

# 알튜비튜 트리

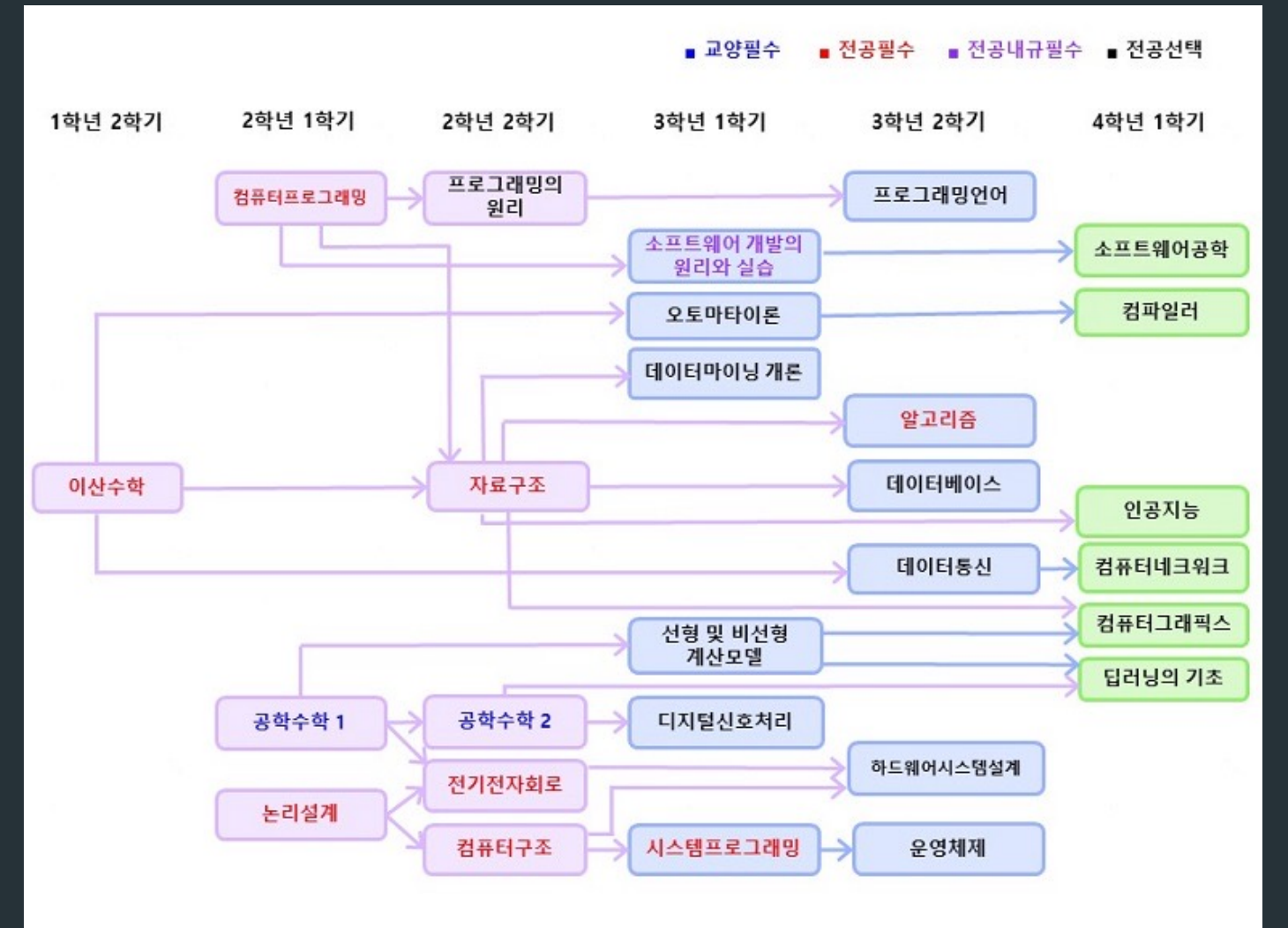
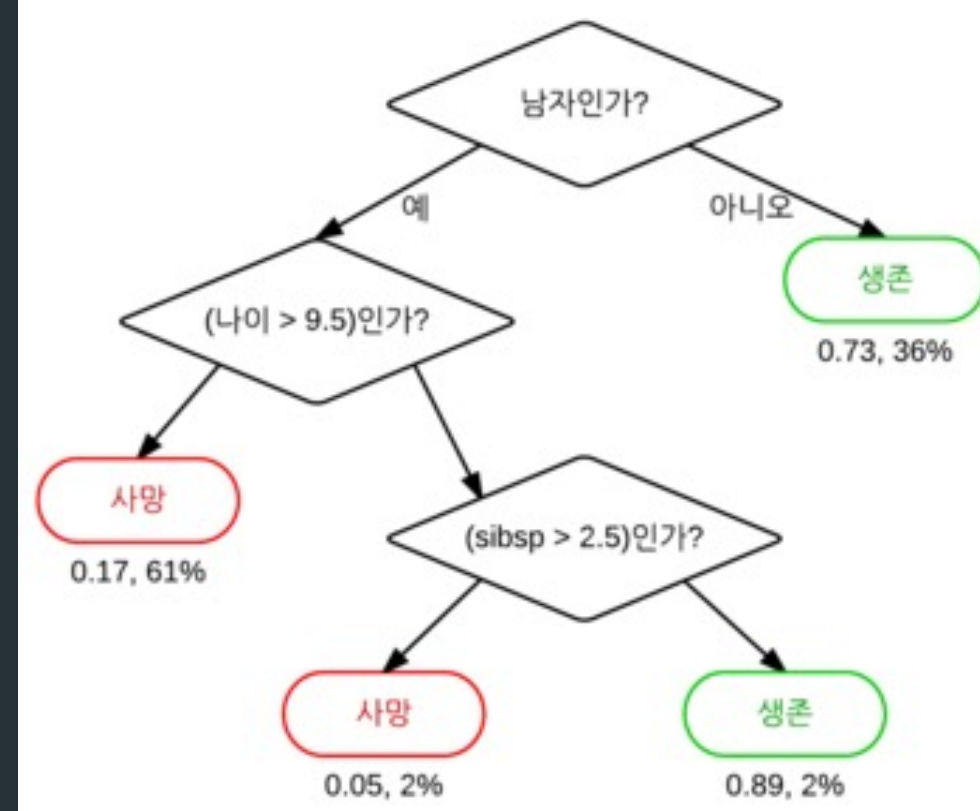


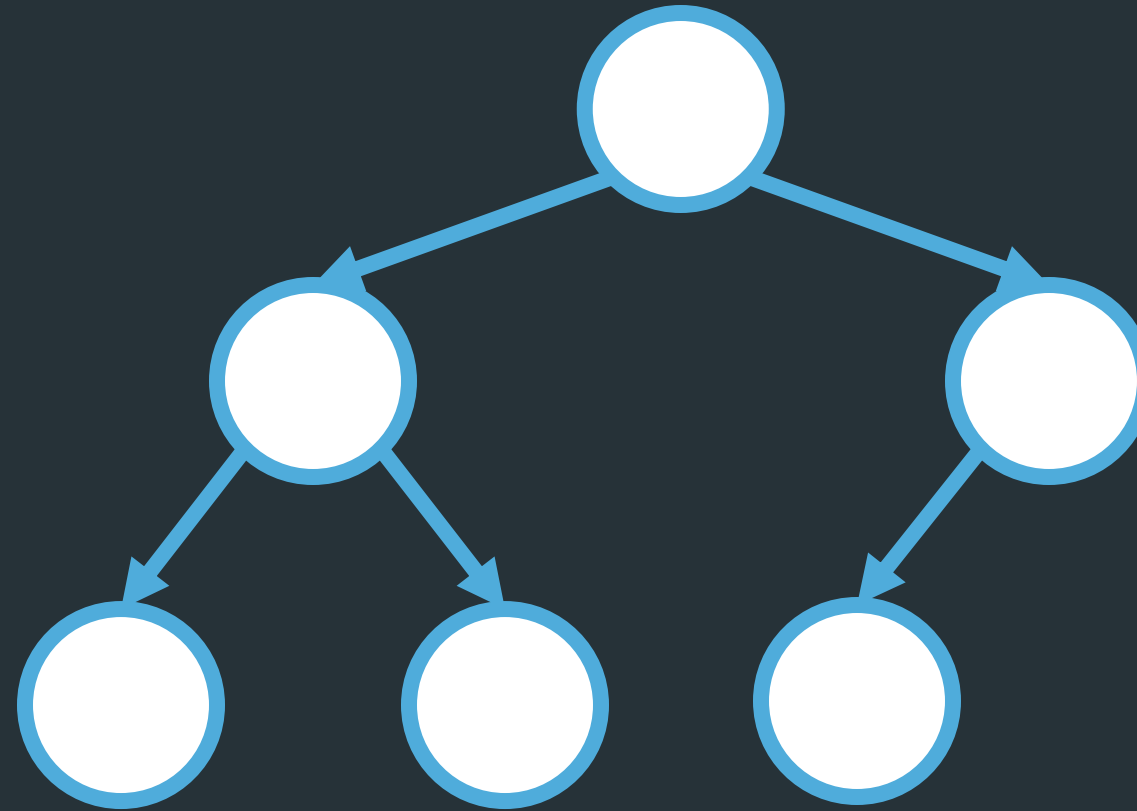
비선형 자료구조 중 하나인 트리입니다.  
그래프의 부분집합으로 계층 관계를 나타낼 때 주로 사용해요.

# 일상 속 트리

```
C:\Users\iw040\Notice (main -> origin)
λ tree
Folder PATH listing
Volume serial number is C0000100 EE98:4106
C:.
```

```
09월 03일 - 정렬
과제
├── .idea
├── cmake-build-debug
│   ├── CMakeFiles
│   │   ├── 3.17.5
│   │   │   ├── CompilerIdC
│   │   │   │   └── tmp
│   │   │   ├── CompilerIdCXX
│   │   │   │   └── tmp
│   │   ├── CMakeTmp
│   │   └── CMAKE_ROOT.dir
│   ├── Testing
│   └── Temporary
라이브 코딩
├── .idea
├── cmake-build-debug
│   ├── CMakeFiles
│   │   ├── 3.17.5
│   │   │   ├── CompilerIdC
│   │   │   │   └── tmp
│   │   │   ├── CompilerIdCXX
│   │   │   │   └── tmp
│   │   ├── CMakeTmp
│   │   └── CMAKE_ROOT.dir
│   ├── .dir
│   ├── Testing
│   └── Temporary
```



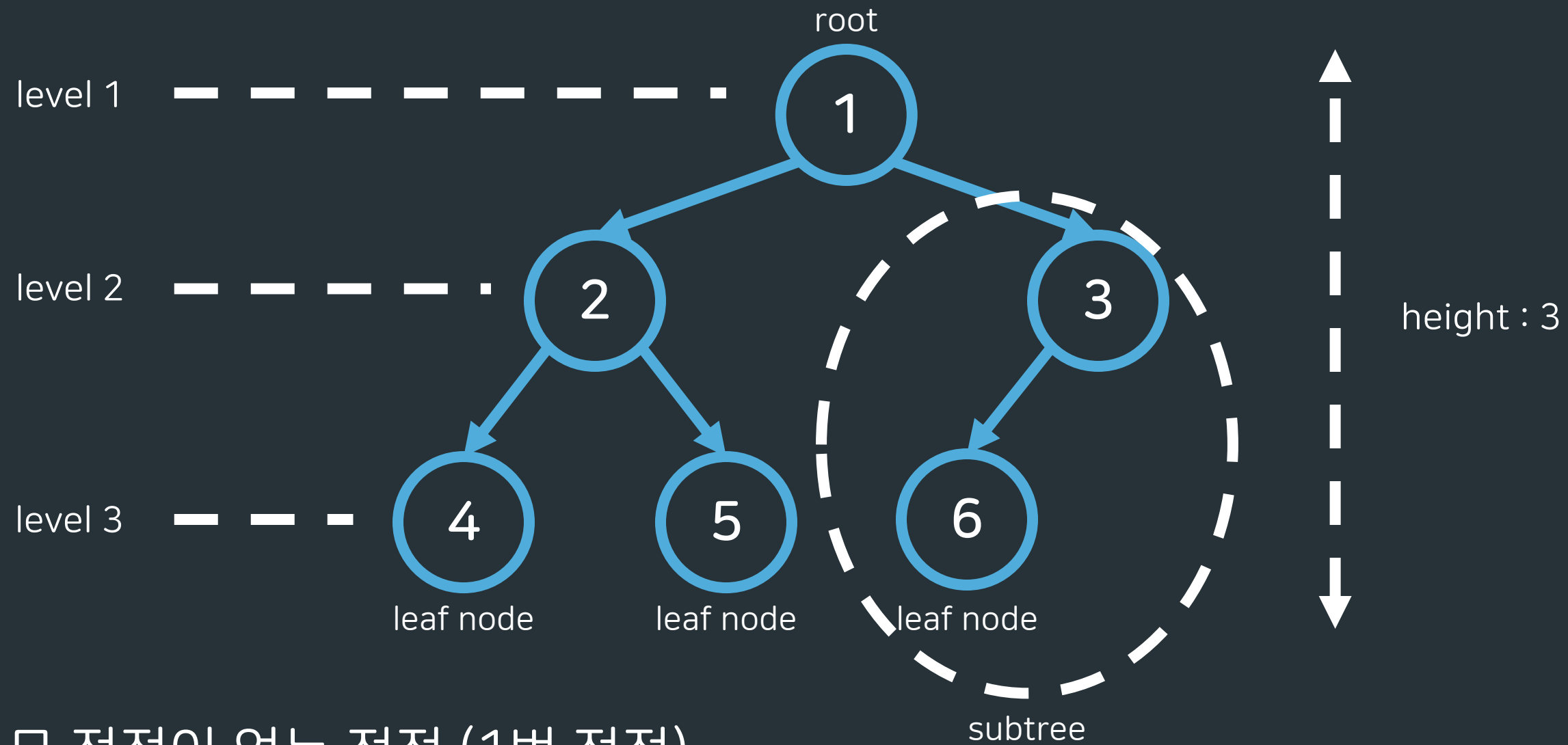


## Tree

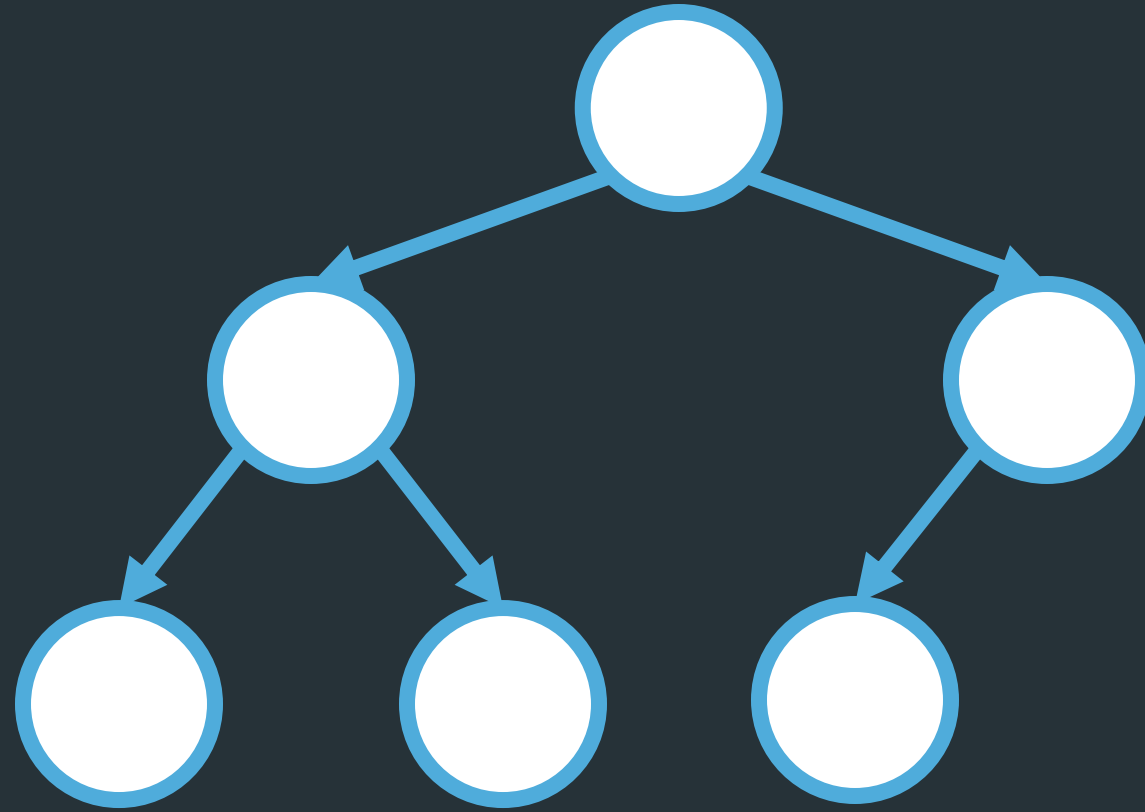
- 비선형 자료구조
- 그래프의 부분집합으로 사이클이 없고,  $V$ 개의 정점에 대해  $V-1$ 개의 간선이 있음
- 부모-자식의 계층 구조
- 트리 탐색의 시간 복잡도는  $O(h)$  ( $h$  = 트리의 높이)
- 그래프와 마찬가지로 DFS, BFS를 이용하여 탐색

	트리	그래프
간선의 수	$V-1$ 개	*
특정 정점 사이 경로의 수	1개	*
방향 유무	0	$\Delta$
사이클 유무	X	$\Delta$
계층 관계	0	X

## Tree

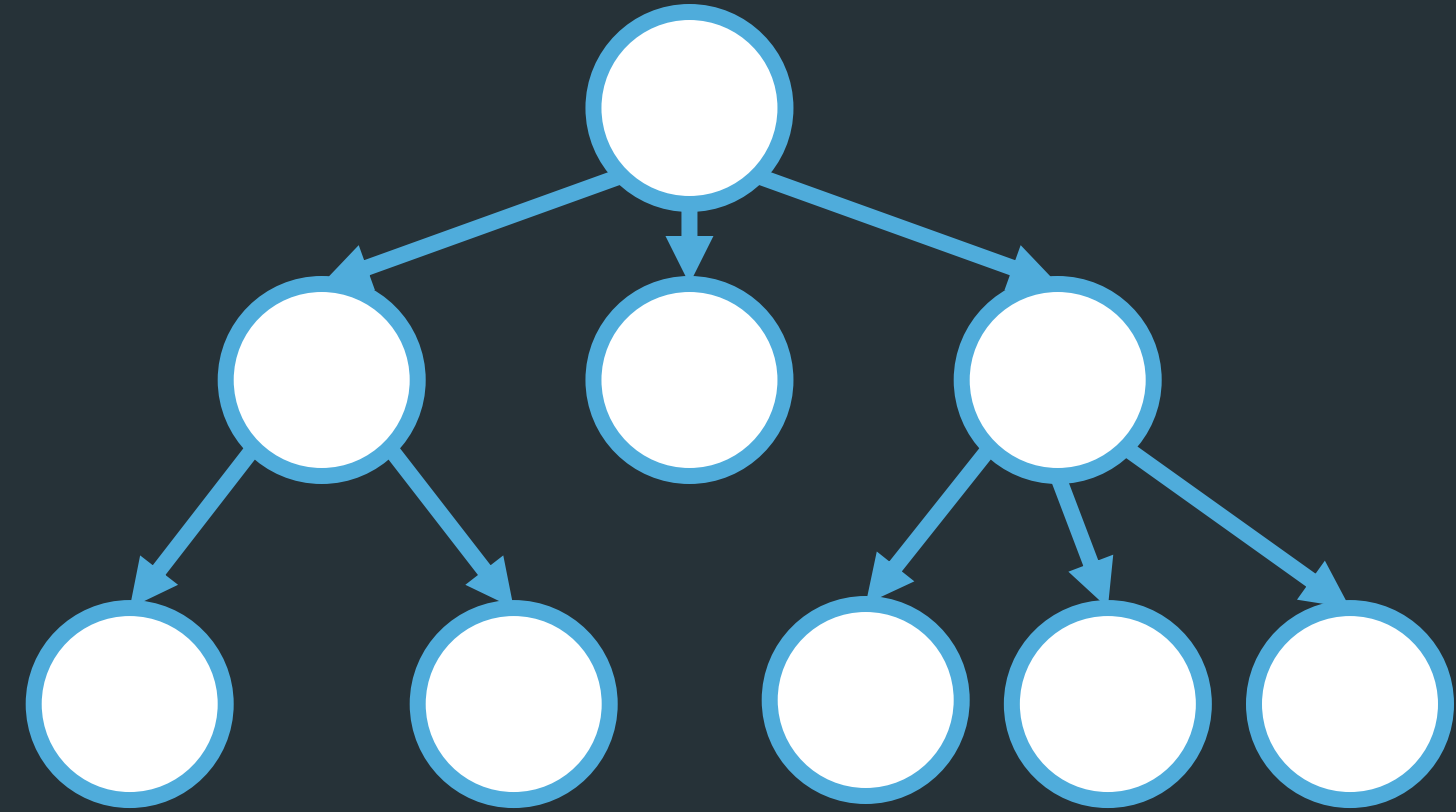


- Root : 부모 정점이 없는 정점 (1번 정점)
- Subtree : 트리의 부분 집합
- Leaf node : 자식 정점이 없는 정점 (4, 5, 6번 정점)
- Level : 트리의 각 계층
- Height : level 중 가장 큰 값



Binary Tree (이진 트리)

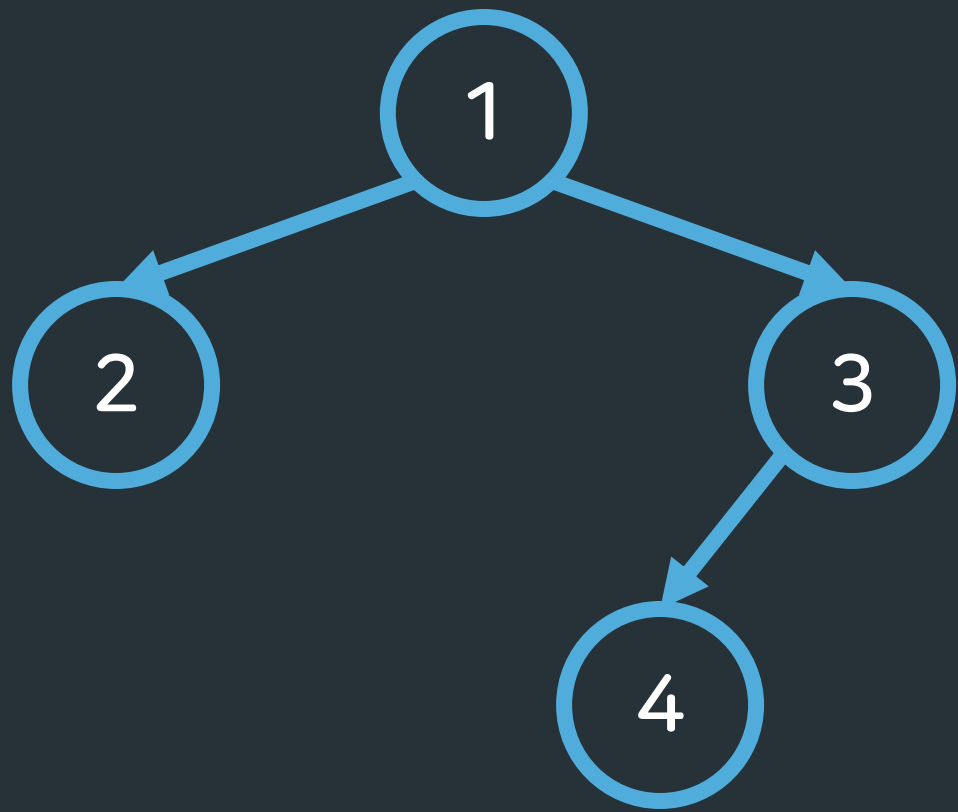
자식 정점의 수가 2개 이하



General Tree (일반 트리)

자식 정점의 수에 제한 없음

# 트리 구현 (이진 트리)



구조체 + 포인터

실시간으로 트리를 만들어야 할 때 적합



```
struct Node {  
    int data;  
    Node *left;  
    Node *right;  
};
```

```
int main() {  
    Node *n1 = new Node();  
    Node *n2 = new Node();  
    Node *n3 = new Node();  
    Node *n4 = new Node();
```

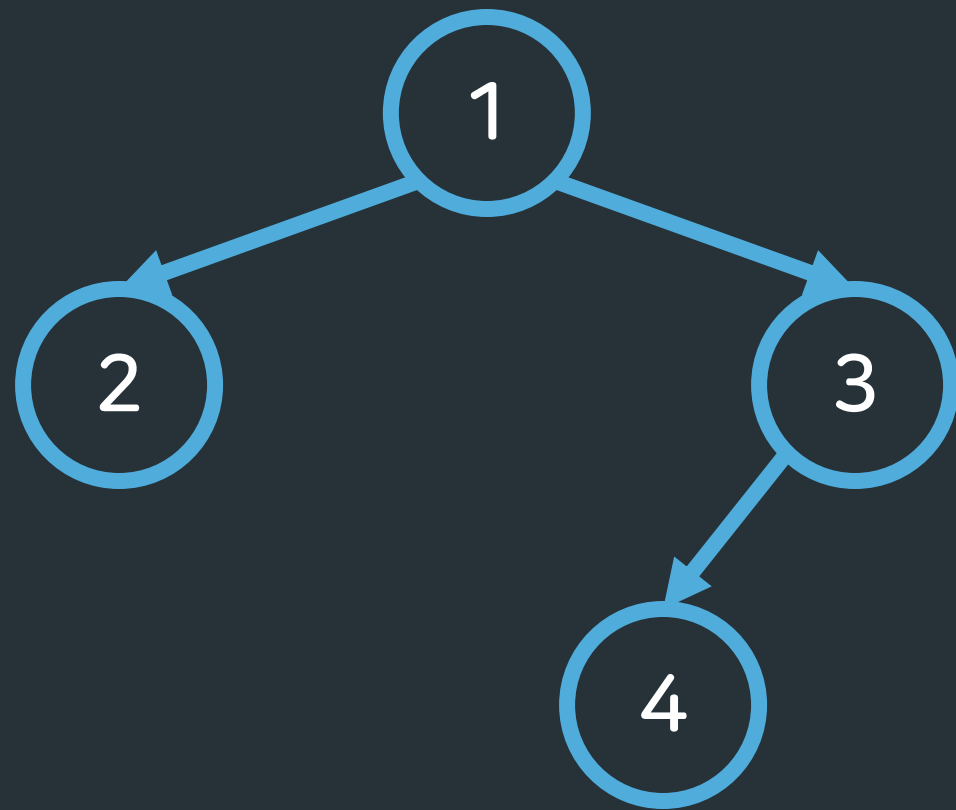
```
n1->data = 1;  
n1->left = n2;  
n1->right = n3;
```

```
n2->data = 2;  
n2->left = n2->right = NULL;
```

```
n3->data = 3;  
n3->left = n4;  
n3->right = NULL;
```

```
n4->data = 4;  
n4->left = n4->right = NULL;  
}
```

# 트리 구현 (이진 트리)



맵

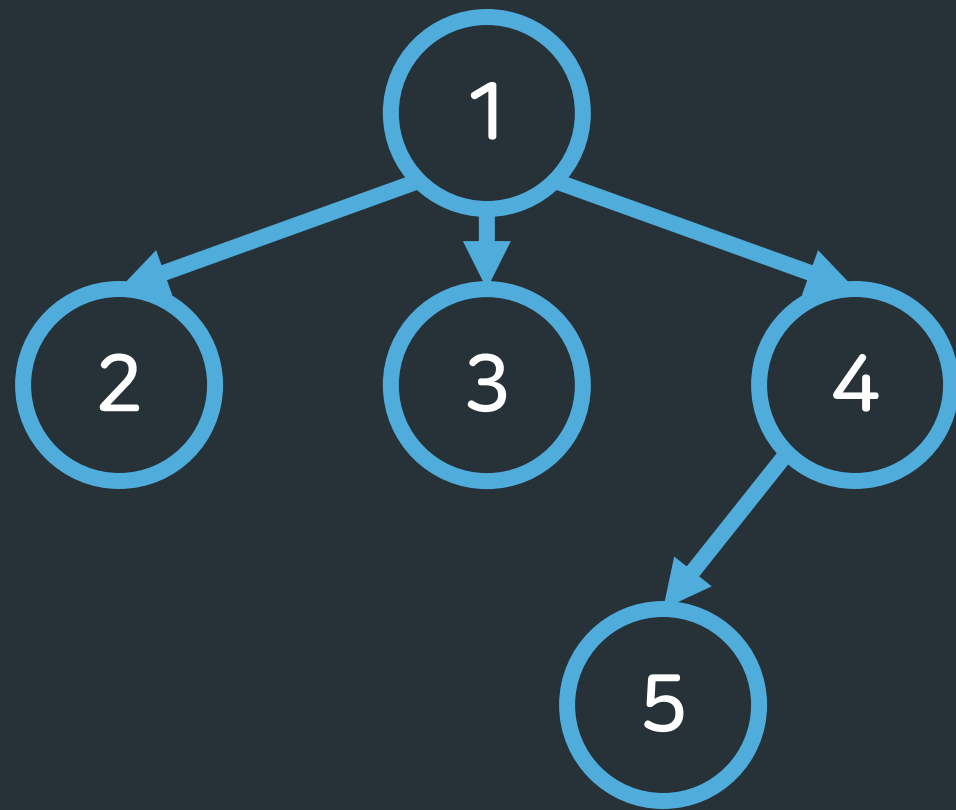
이미 트리 관계가 정의되어 있을 때 적합



```
int main() {  
    map<int, pair<int, int>> tree;  
  
    tree[1] = {2, 3};  
    tree[2] = {-1, -1};  
    tree[3] = {4, -1};  
    tree[4] = {-1, -1};  
}
```



# 트리 구현 (일반 트리)



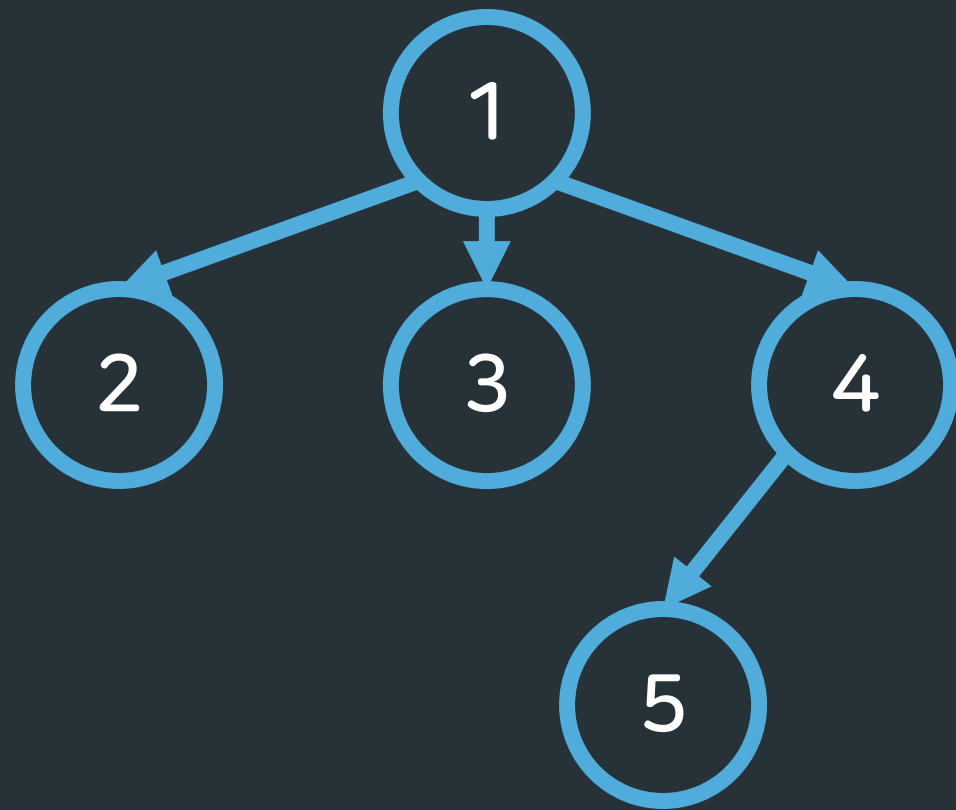
맵

정점 번호가 연속하지 않을 때 적합



```
int main() {  
    map<int, vector<int>> tree;  
  
    tree[1] = {2, 3, 4};  
    tree[4] = {5};  
}
```

# 트리 구현 (일반 트리)

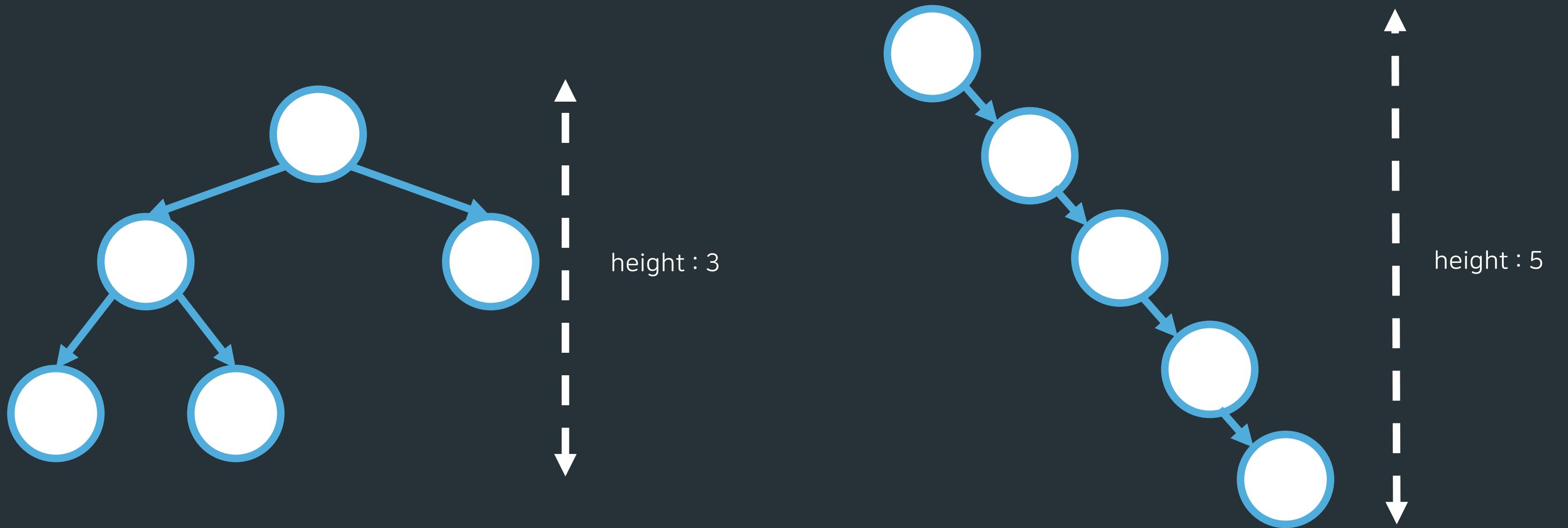


2차원 벡터

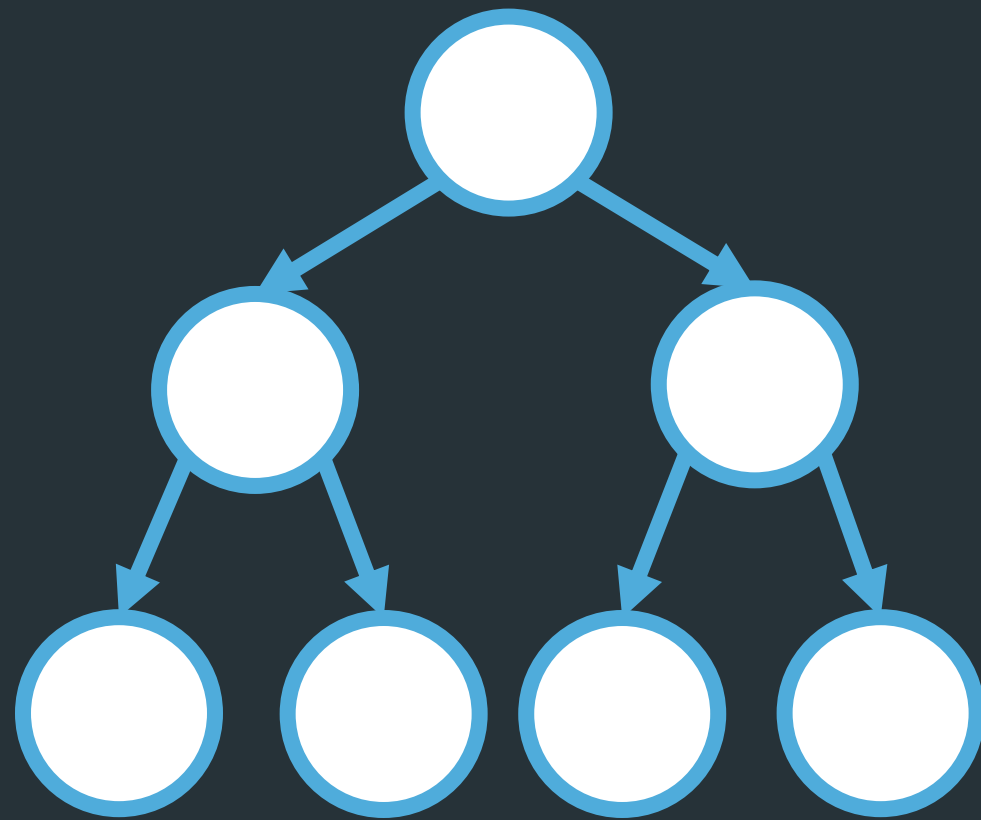
정점 번호가 연속할 때 적합



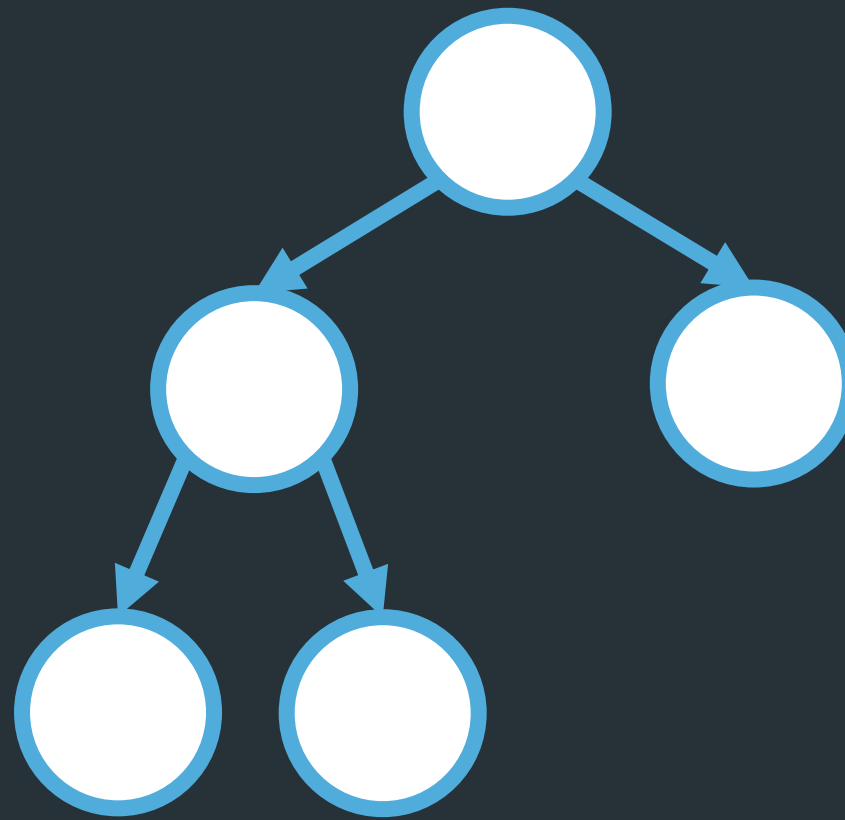
```
int main() {  
    vector<vector<int>> tree;  
  
    tree[1] = {2, 3, 4};  
    tree[4] = {5};  
}
```



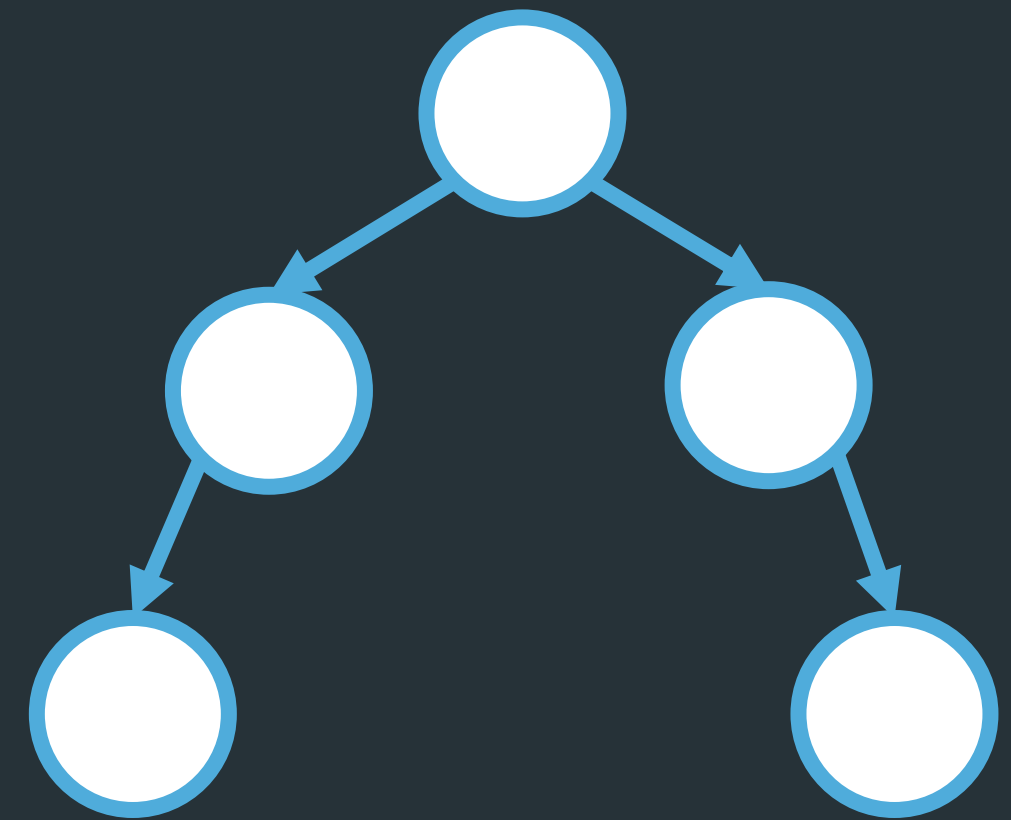
$(\text{ceil}) \log(V+1) \leq \text{이진 트리의 높이} \leq V$   
 $O((\text{ceil}) \log(V+1)) \leq \text{이진 트리의 시간 복잡도} \leq O(V)$



Full Binary Tree

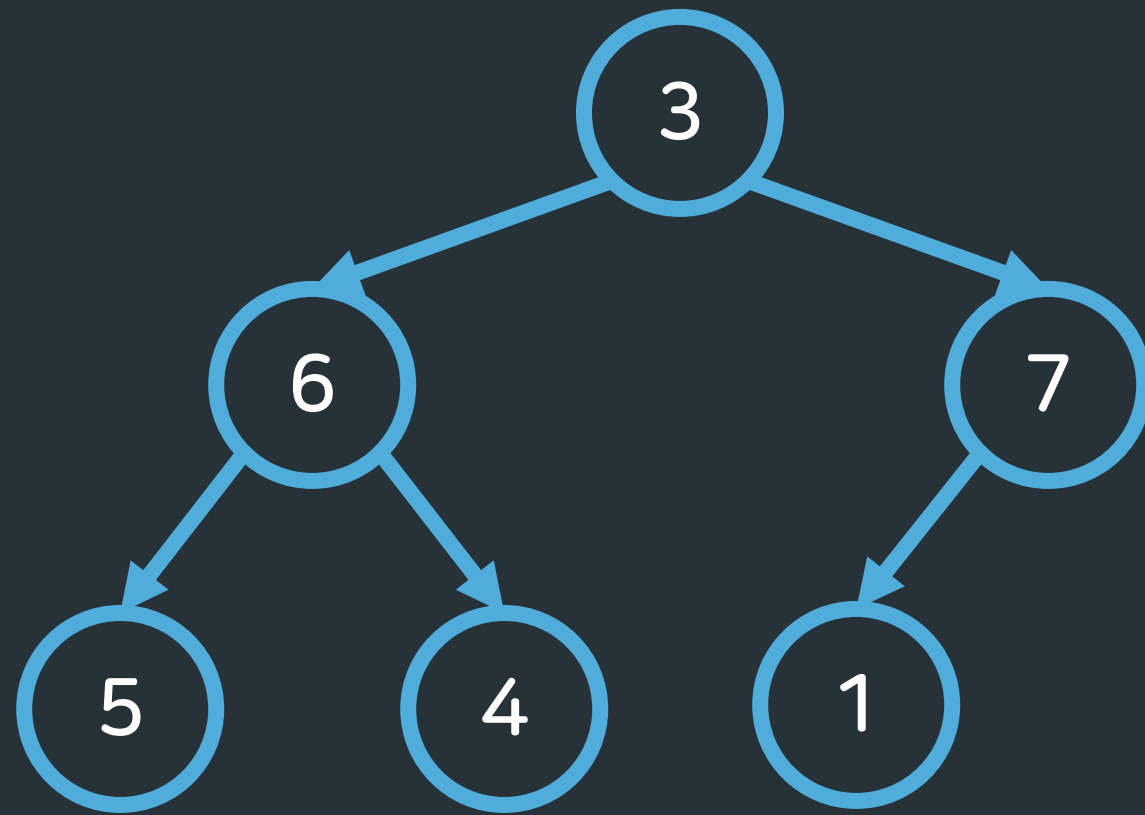


Complete Binary Tree



Other Binary Tree

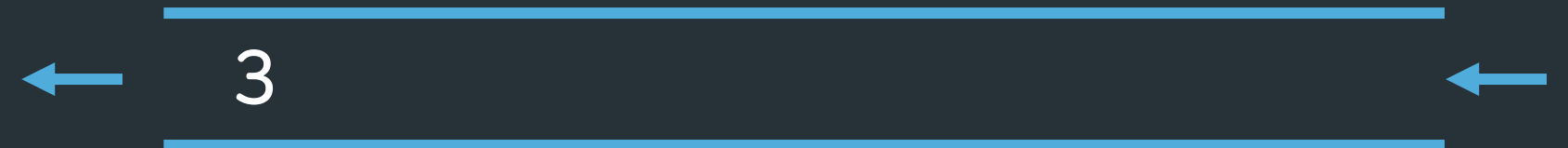
레벨 순회 (Level traversal)  
전위 순회 (Preorder traversal)  
중위 순회 (Inorder traversal)  
후위 순회 (Postorder traversal)

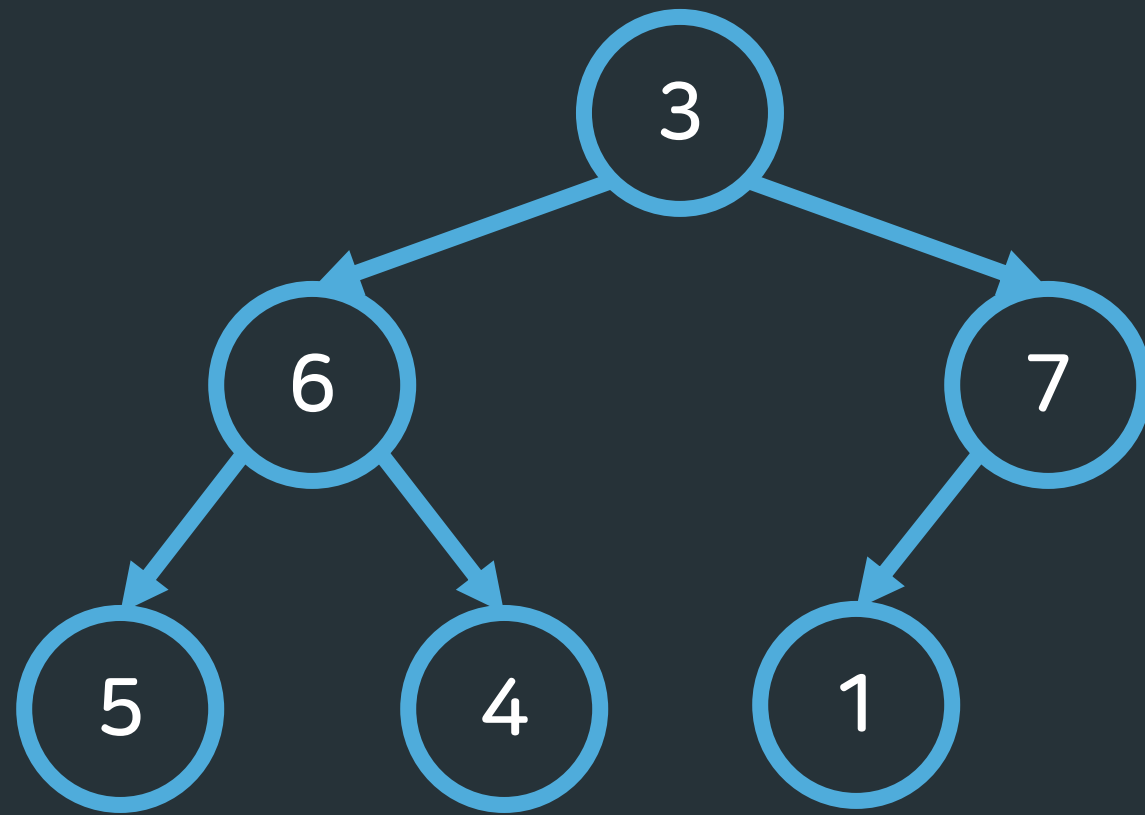


- 탐색 순서:



```
level (root)
{
    while (!q.empty())
        v = q.front();
        q.pop();
        if (v == null)
            continue;
        q.push(left(v));
        q.push(right(v));
}
```

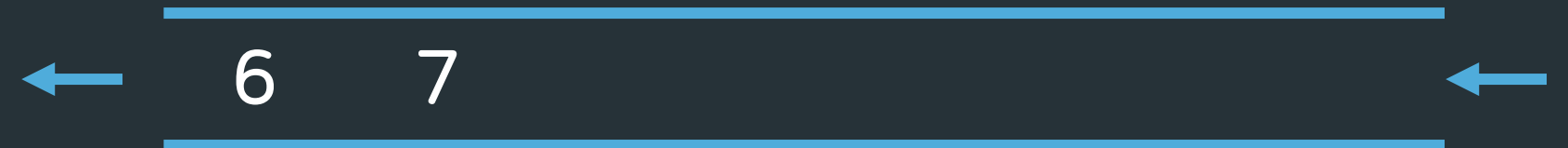


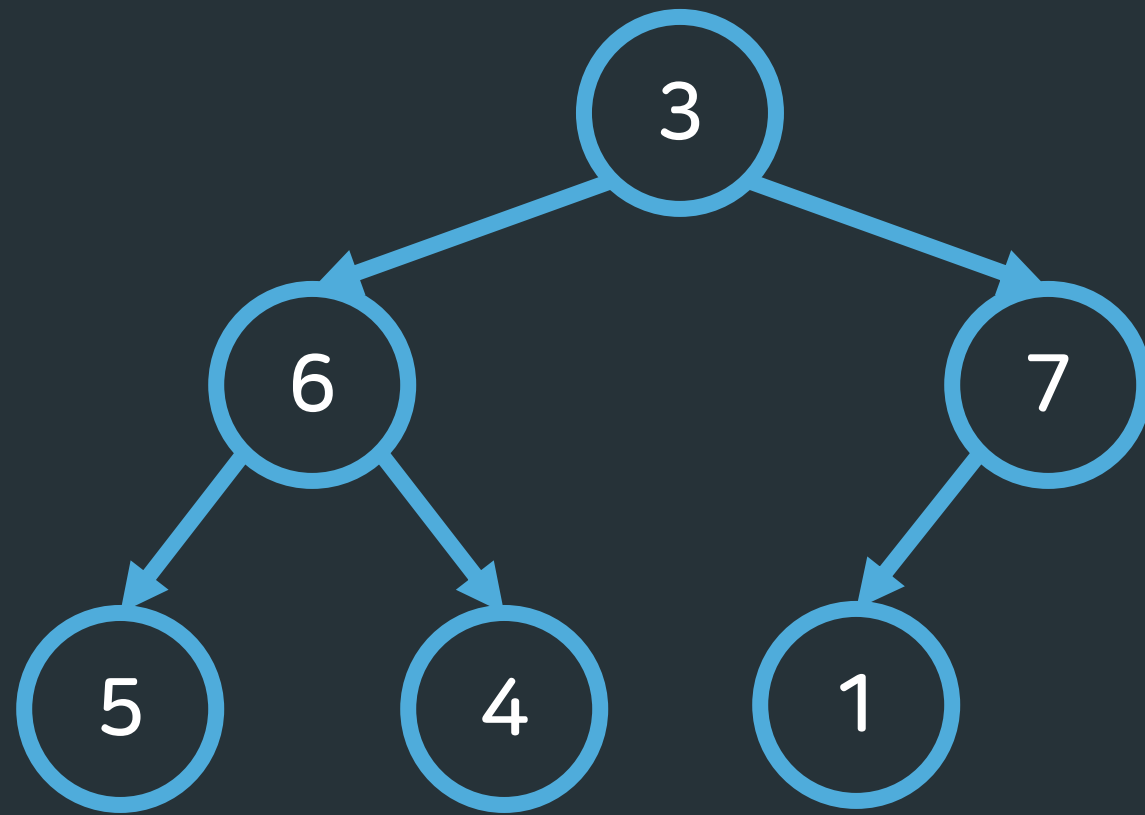


- 탐색 순서: 3



```
level (root)
{
    while (!q.empty())
        v = q.front();
        q.pop();
        if (v == null)
            continue;
        q.push(left(v));
        q.push(right(v));
}
```

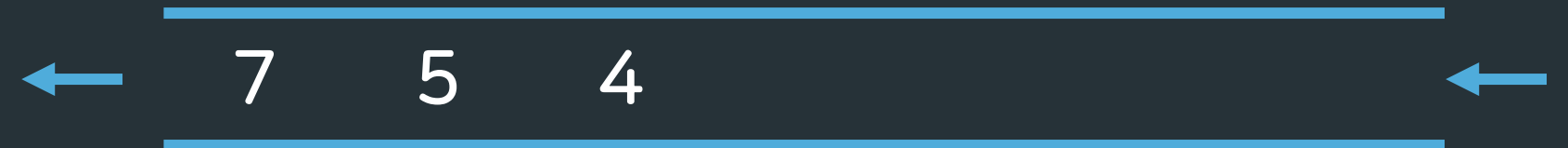




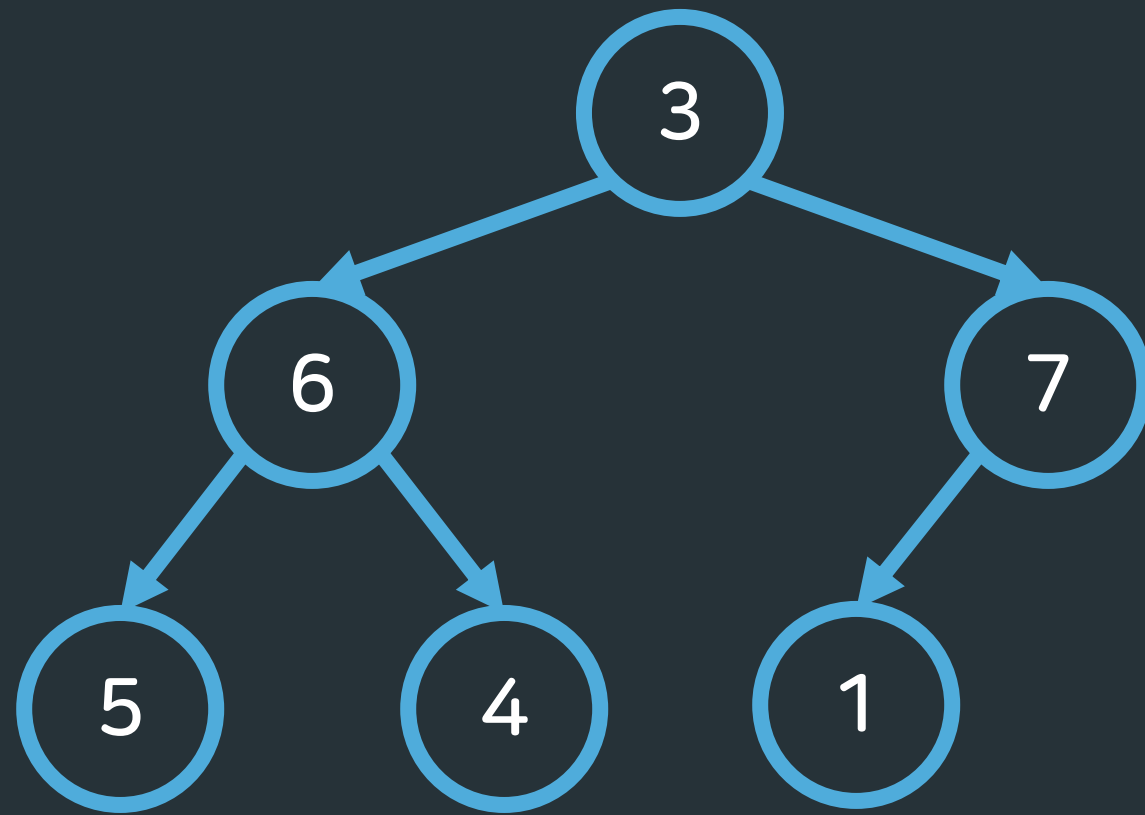
- 탐색 순서: 3 → 6



```
level (root)
{
    while (!q.empty())
        v = q.front();
        q.pop();
        if (v == null)
            continue;
        q.push(left(v));
        q.push(right(v));
}
```



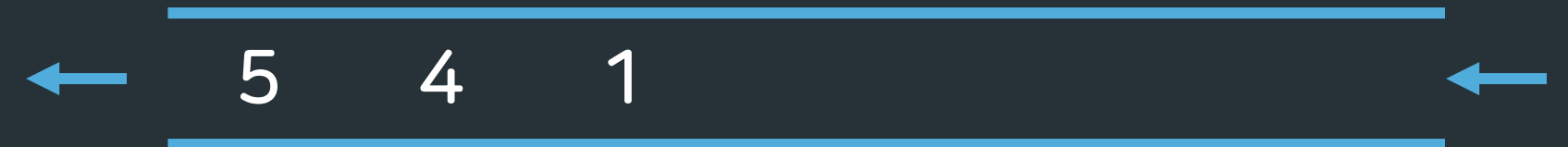




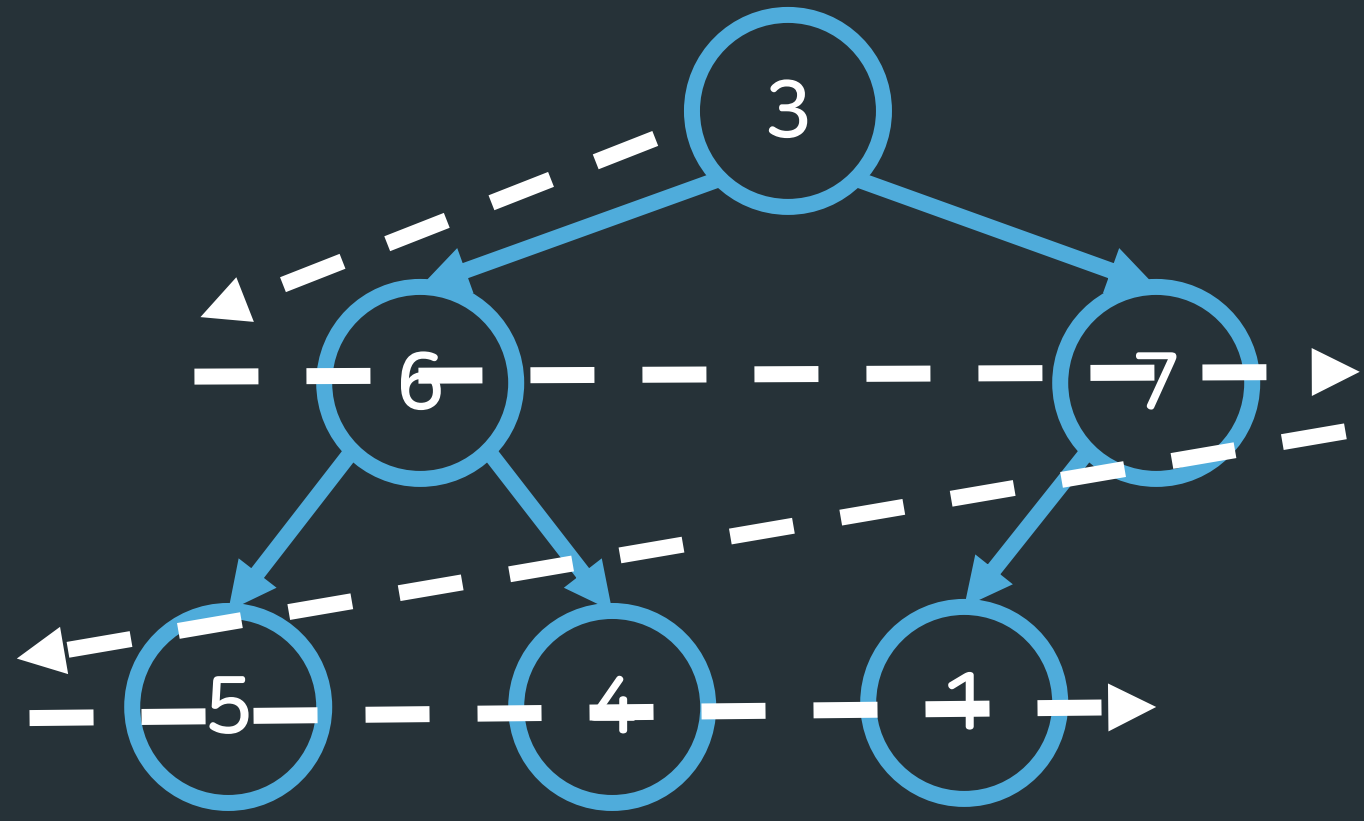
- 탐색 순서: 3 → 6 → 7



```
level (root)
{
    while (!q.empty())
        v = q.front();
        q.pop();
        if (v == null)
            continue;
        q.push(left(v));
        q.push(right(v));
}
```



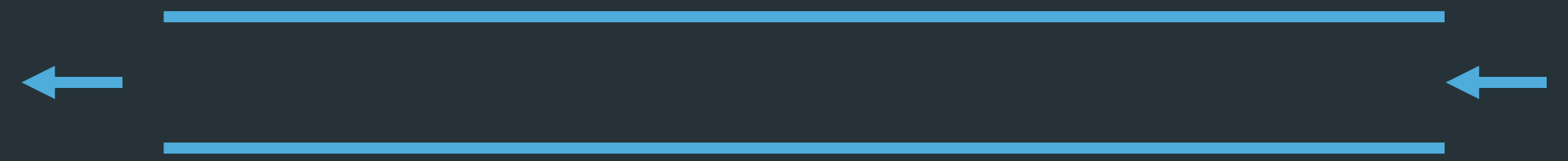
\*일반 트리도 가능

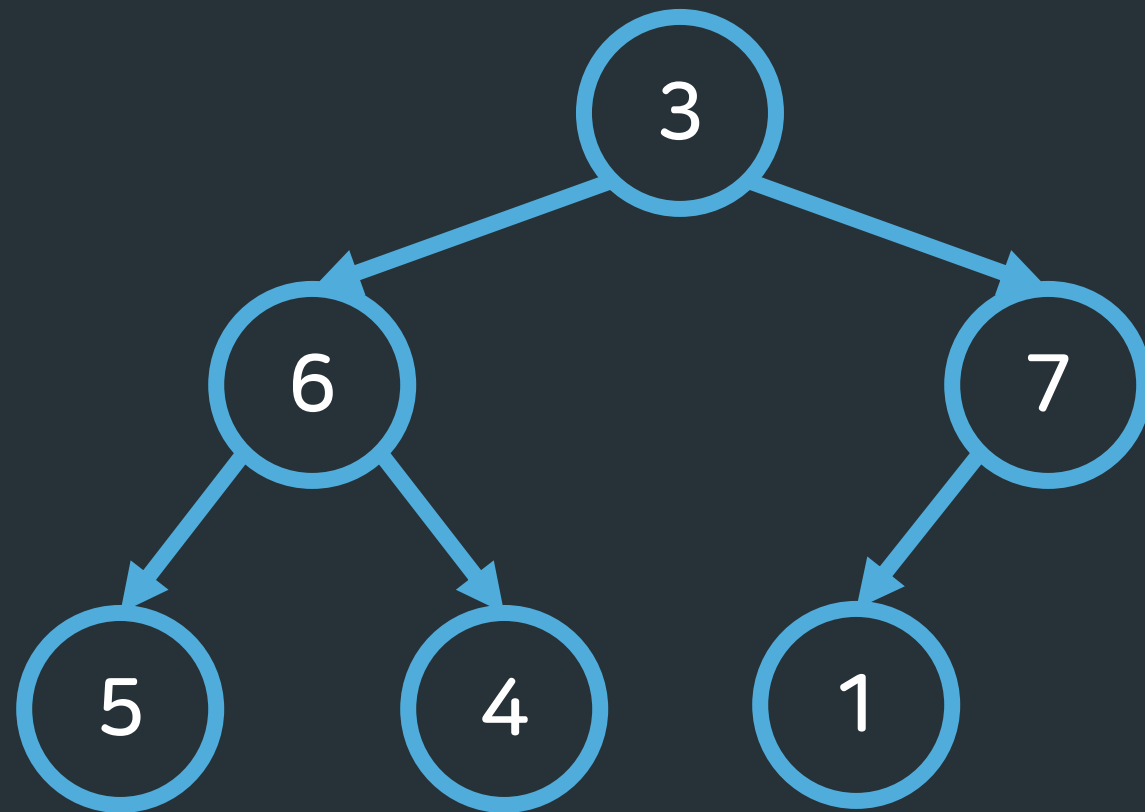


● 탐색 순서: 3 → 6 → 7 → 5 → 4 → 1



```
level (root)
{
    while (!q.empty())
        v = q.front();
        q.pop();
        if (v == null)
            continue;
        q.push(left(v));
        q.push(right(v));
}
```

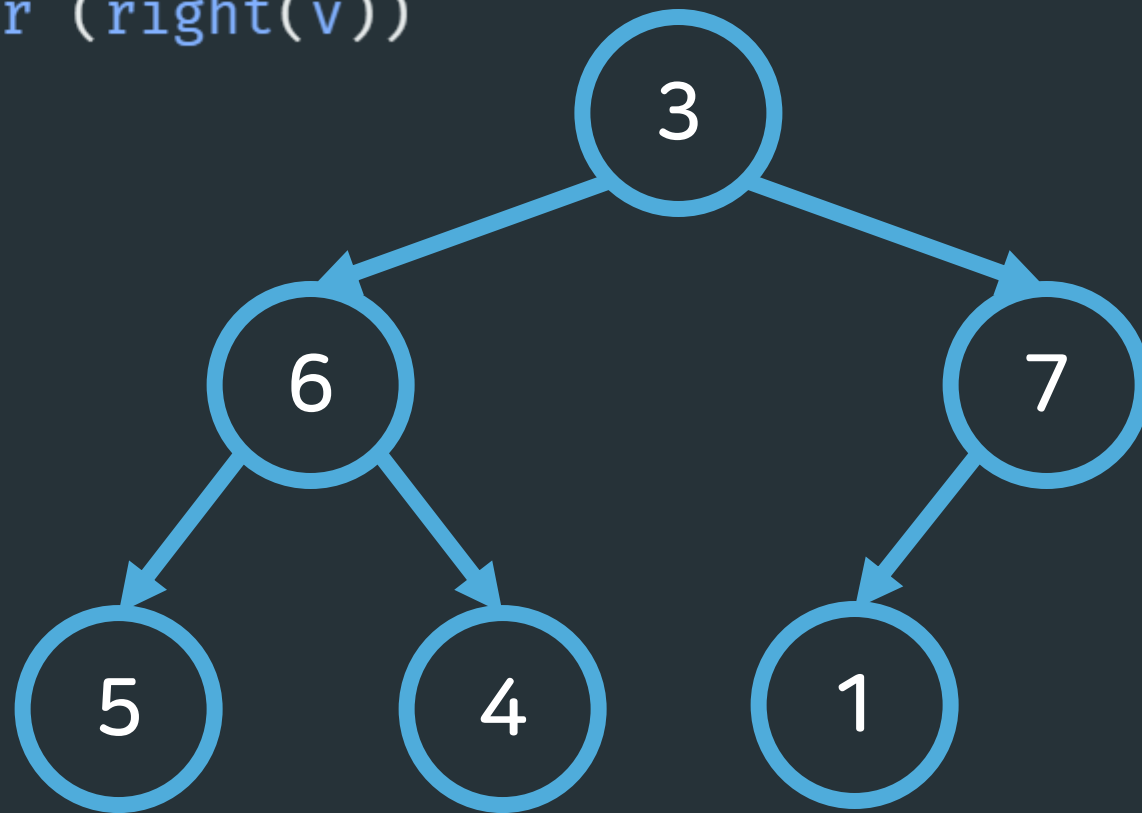




```
preorder (v)
{
    if (v ≠ null)
        print (v)
        preorder (left(v))
        preorder (right(v))
}
```

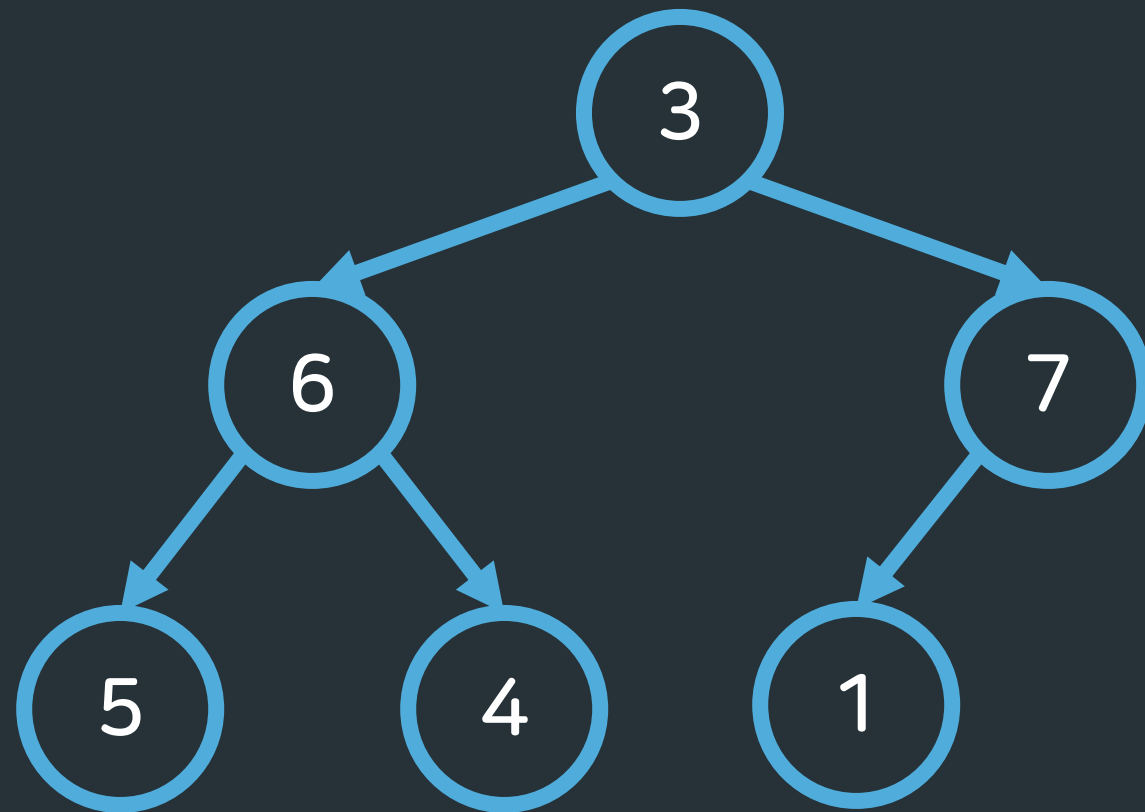


```
preorder (v)
{
    if (v ≠ null)
        print (v)
        preorder (left(v))
        preorder (right(v))
}
```



3 6 5 4 7 1

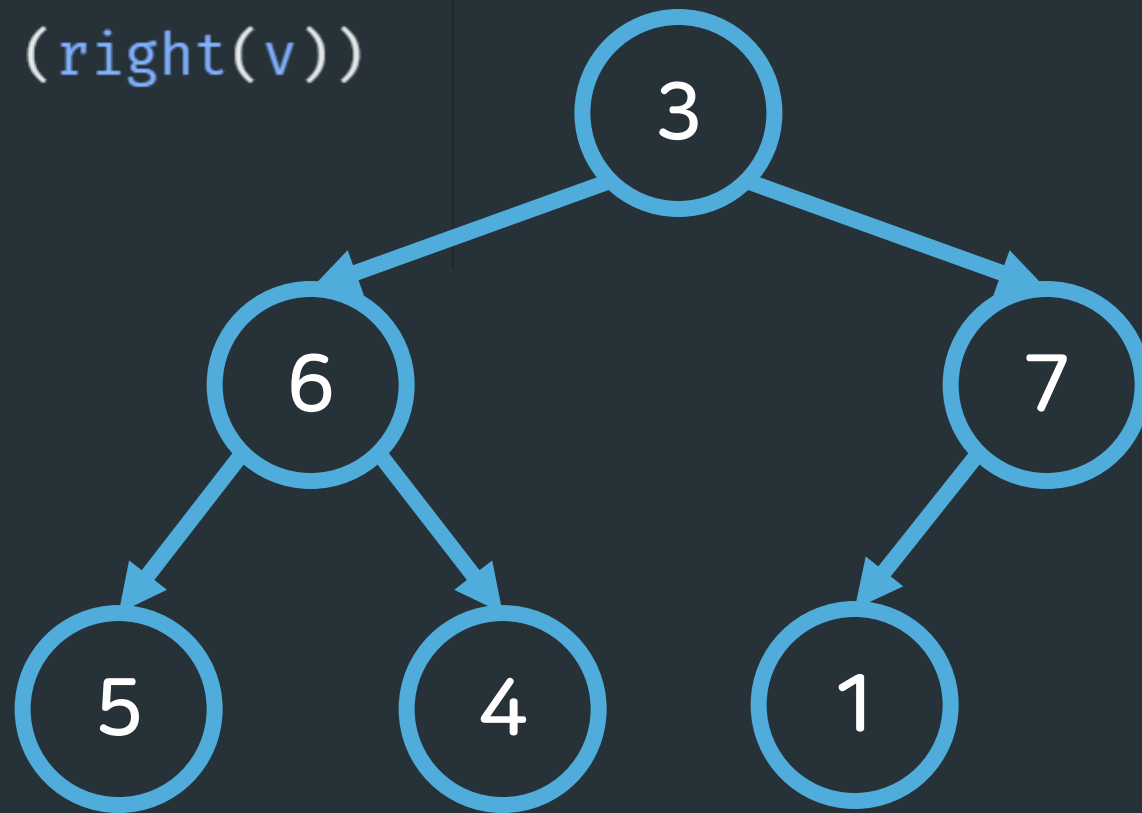
```
preorder(3)
  print(3)
  preorder(3 -> left : 6)
    print(6)
    preorder(6 -> left : 5)
      print(5)
      preorder(5 -> left : null)
      preorder(5 -> right : null)
    preorder(6 -> right : 4)
      print(4)
      preorder(4 -> left : null)
      preorder(4 -> right : null)
  preorder(3 -> right : 7)
    print(7)
    preorder(7 -> left : 1)
      print(1)
      preorder(1 -> left : null)
      preorder(1 -> right : null)
    preorder(7 -> right : null)
```



```
inorder (v)
{
    if (v ≠ null)
        inorder (left(v))
        print (v)
        inorder (right(v))
}
```

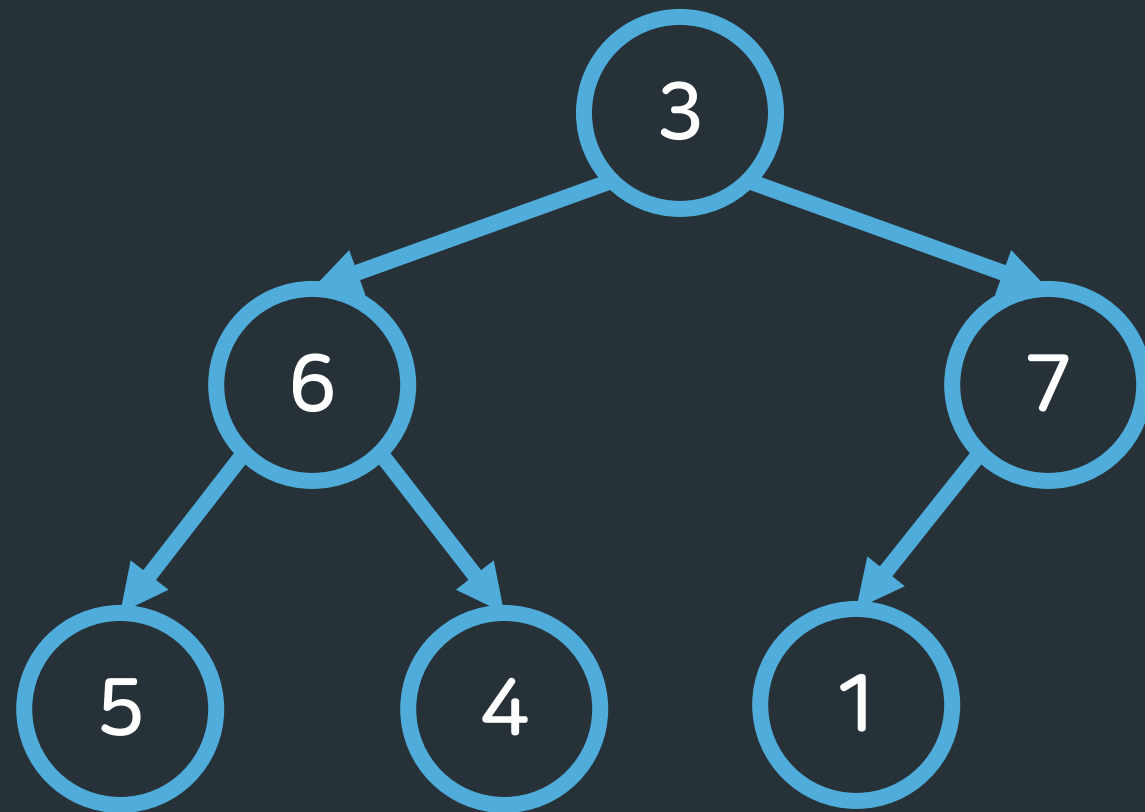


```
inorder (v)
{
    if (v ≠ null)
        inorder (left(v))
        print (v)
        inorder (right(v))
}
```



5 6 4 3 1 7

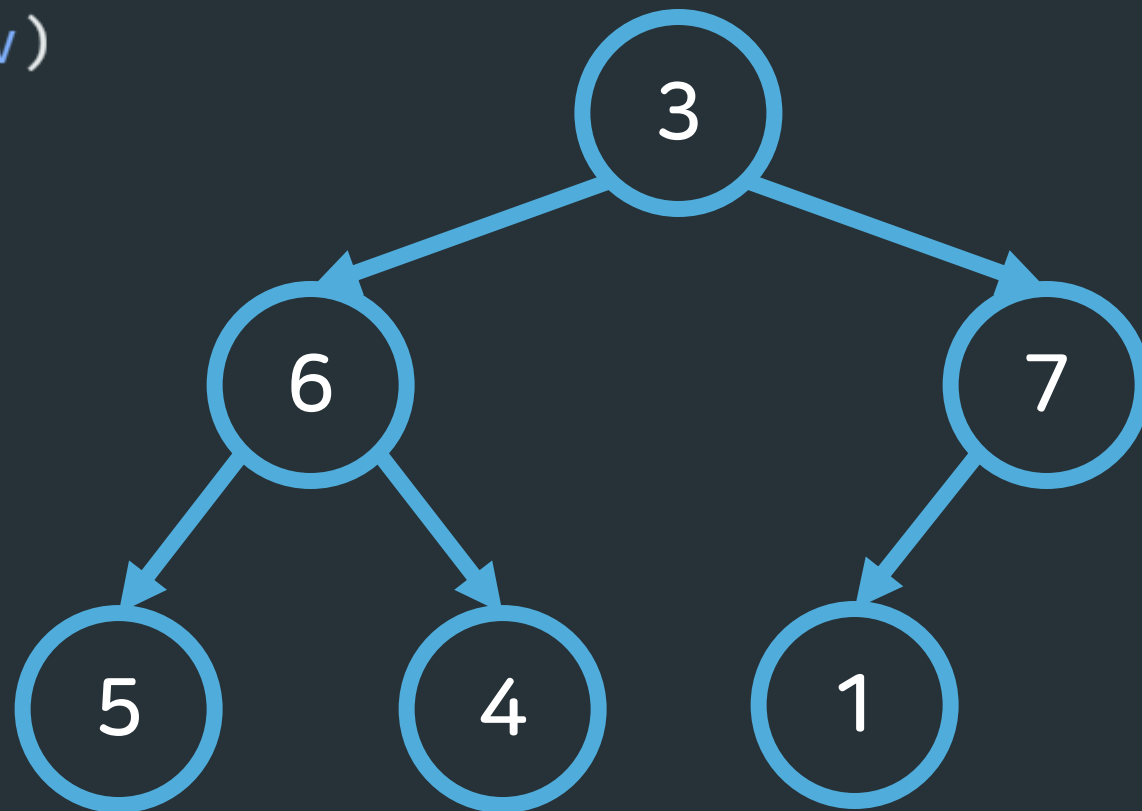
```
inorder(3)
  inorder(3 -> left : 6)
    inorder(6 -> left : 5)
      inorder(5 -> left : null)
      print(5)
      inorder(5 -> right : null)
    print(6)
    inorder(6 -> right : 4)
      inorder(4 -> left : null)
      print(4)
      inorder(4 -> right : null)
  print(3)
  inorder(3 -> right : 7)
    inorder(7 -> left : 1)
      inorder(1 -> left : null)
      print(1)
      inorder(1 -> right : null)
    print(7)
    inorder(7 -> right : null)
```



```
postorder (v)
{
    if (v ≠ null)
        postorder (left(v))
        postorder (right(v))
        print (v)
}
```



```
postorder (v)
{
    if (v ≠ null)
        postorder (left(v))
        postorder (right(v))
        print (v)
}
```



5 4 6 1 7 3

```
postorder(3)
  postorder(3 -> left : 6)
    postorder(6 -> left : 5)
      postorder(5 -> left : null)
      postorder(5 -> right : null)
      print(5)
    postorder(6 -> right : 4)
      postorder(4 -> left : null)
      postorder(4 -> right : null)
      print(4)
    print(6)
  postorder(3 -> right : 7)
    postorder(7 -> left : 1)
      postorder(1 -> left : null)
      postorder(1 -> right : null)
      print(1)
    postorder(7 -> right : null)
    print(7)
  print(3)
```



## /<> 1991번 : 트리 순회 - Silver 1

### 문제

- 이진 트리의 전위 순회, 중위 순회, 후위 순회 결과를 출력하라

### 제한 사항

- 정점의 개수  $N$ 은  $1 \leq N \leq 26$
- 모든 정점은 알파벳 대문자
- 루트 정점은 항상 A

## 예제 입력 1

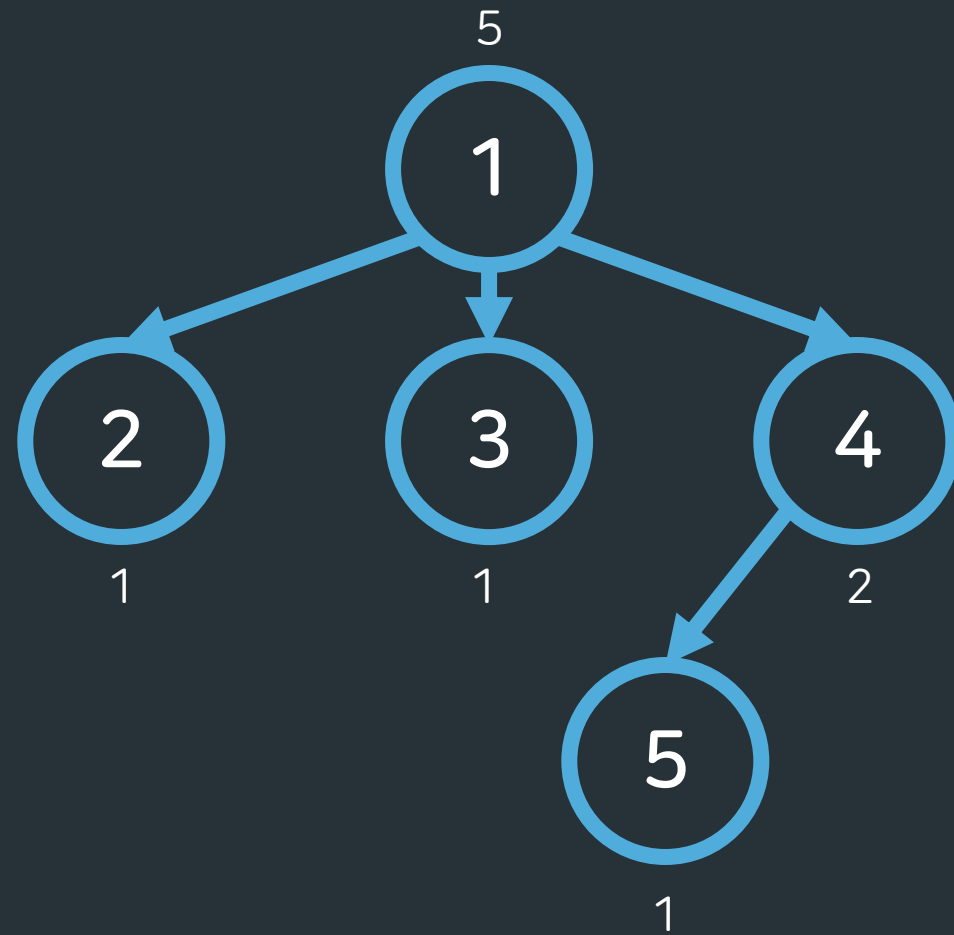
```
7
A B C
B D .
C E F
E ..
F . G
D ..
G ..
```

## 예제 출력 1

```
ABDCEFG
DBAECFG
DBEGFCA
```

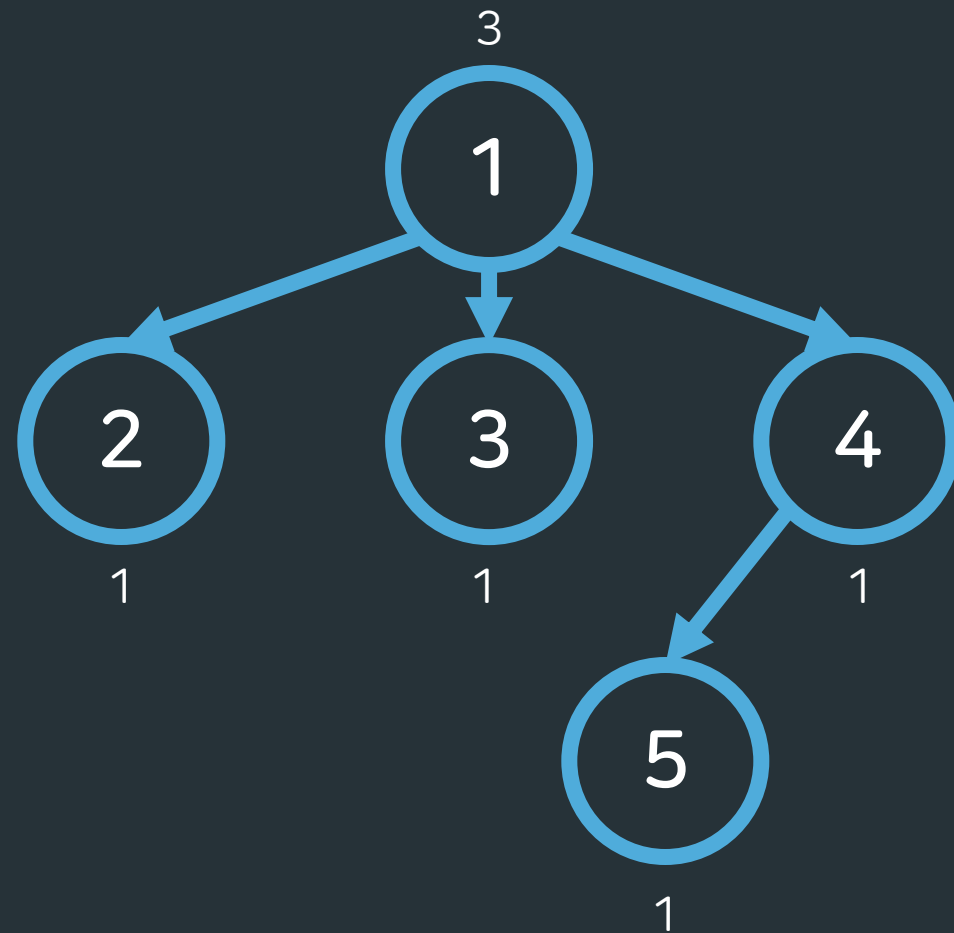
트리의 정점 수 구하기  
리프 노드의 수 구하기  
트리의 높이 구하기

# 트리의 정점 수 구하기



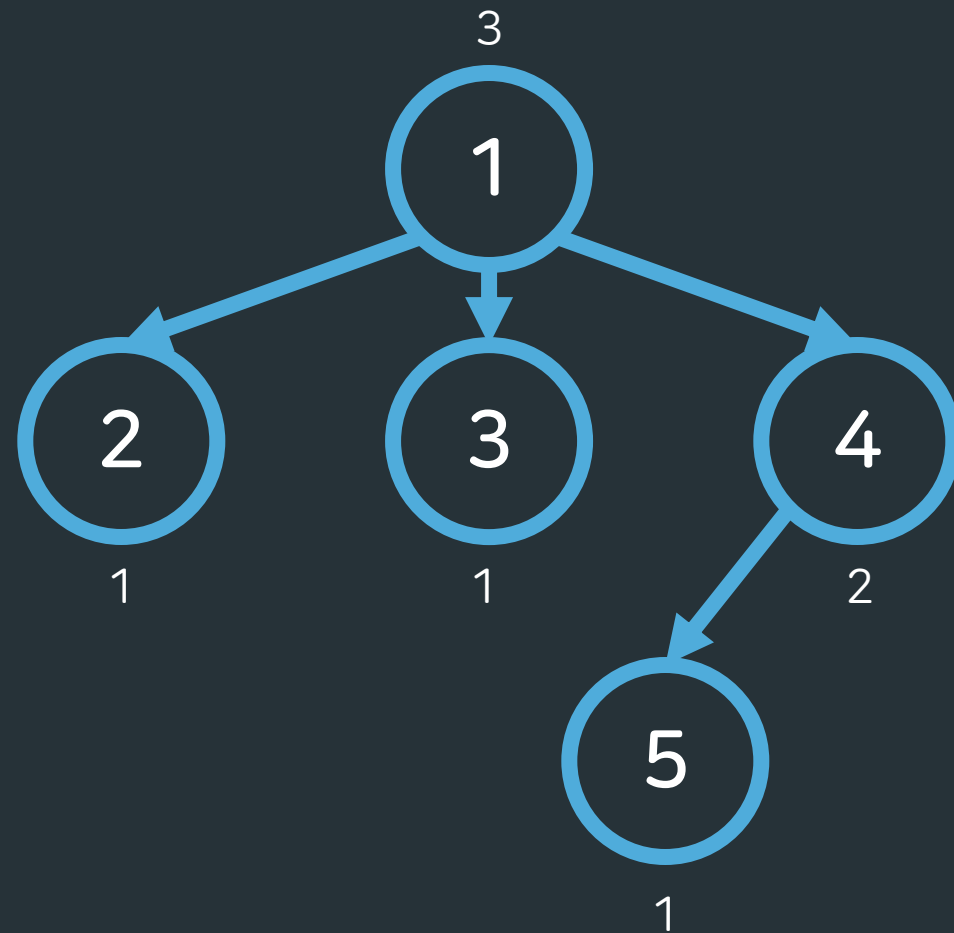
```
int nodeCnt(int node){  
    int cnt = 1;  
    for(int child : tree[node])  
        cnt += nodeCnt(child);  
    return cnt;  
}
```

# 리프 노드의 수 구하기



```
int leafCnt(int node){  
    if(tree[node].empty())  
        return 1;  
  
    int cnt = 0;  
    for(int child : tree[node])  
        cnt += leafCnt(child);  
    return cnt;  
}
```

# 트리의 높이 구하기



```
int treeHeight(int node){  
    int height = 0;  
    for(int child : tree[node])  
        height = max(height, treeHeight(child));  
    return height + 1;  
}
```

## /<> 4803번 : 트리 - Gold 4

### 문제

- 그래프가 주어질 때, **트리의 개수**를 출력하라

### 제한 사항

- 정점의 개수  $n$ 은  $0 \leq n \leq 500$
- 간선의 개수  $m$ 은  $0 \leq m \leq n(n-1)/2$
- 입력은 **무방향 그래프**

## 예제 입력 1

```
6 3
1 2 2 3 3 4
6 5
1 2 2 3 3 4 4 5 5 6
6 6
1 2 2 3 1 3 4 5 5 6 6 4
0 0
```

- 보기 편하게 줄바꿈을 수정했습니다.
- 정확한 입력은 문제 원본을 참고해주세요.

## 예제 출력 1

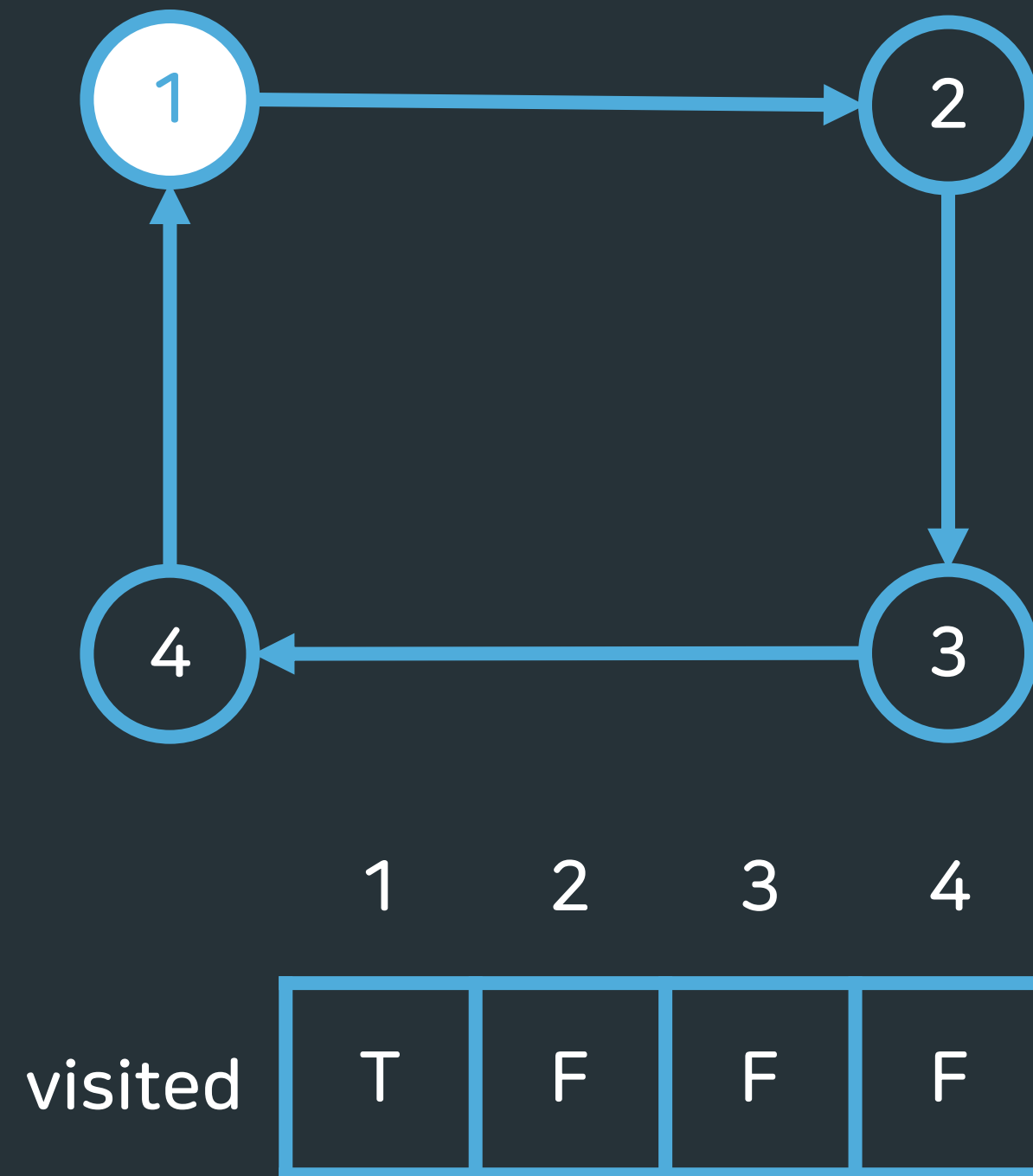
Case 1: A forest of 3 trees.  
Case 2: There is one tree.  
Case 3: No trees.



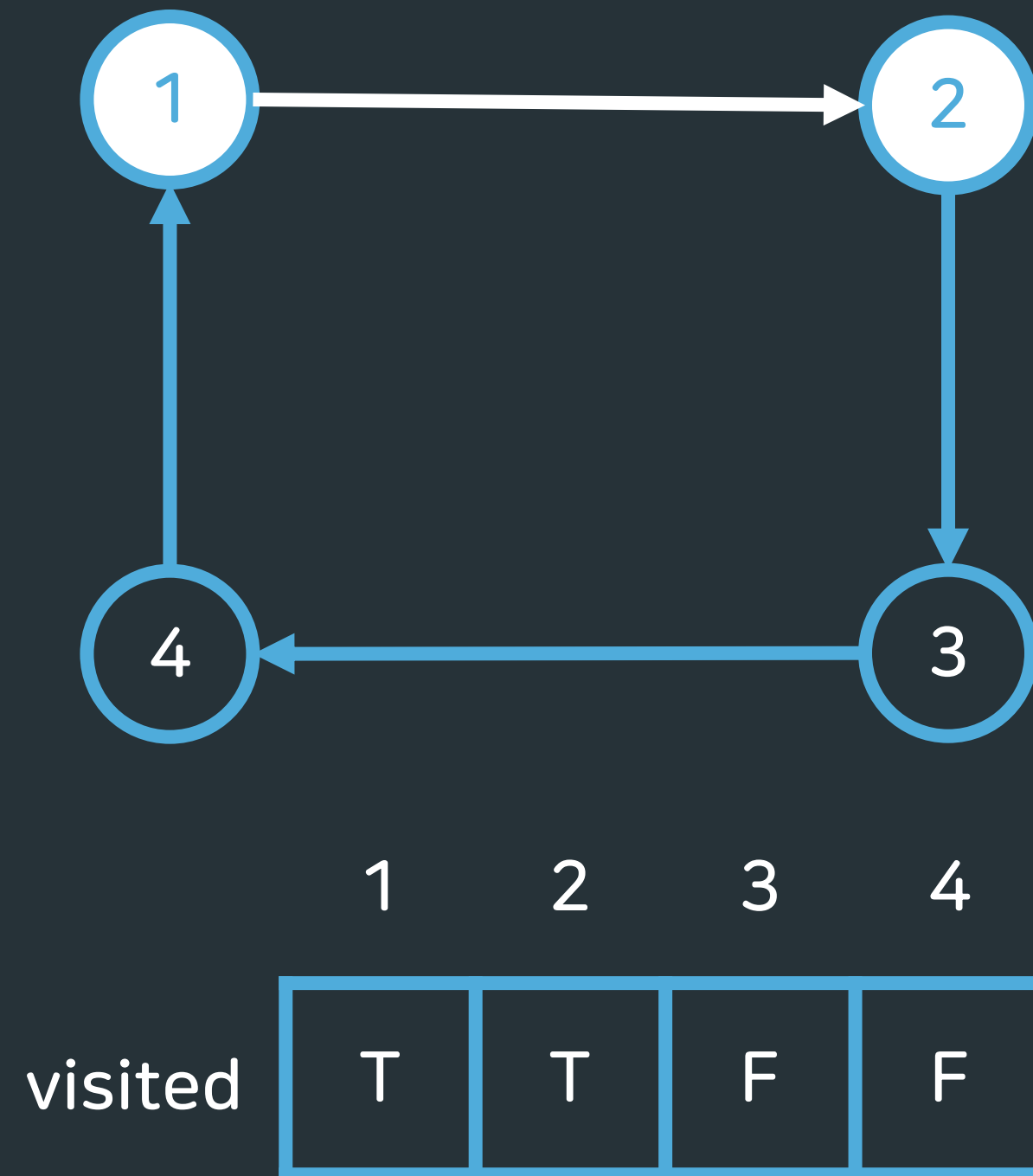
## Hint

1. 사이클 여부를 판단하는 게 중요해요.
2. 사이클이 생성되는 순간을 그려보세요.
3. 트리 순회에는 DFS와 BFS를 사용한다고 했어요. 여기서 뭘 써야 유리 할까요?

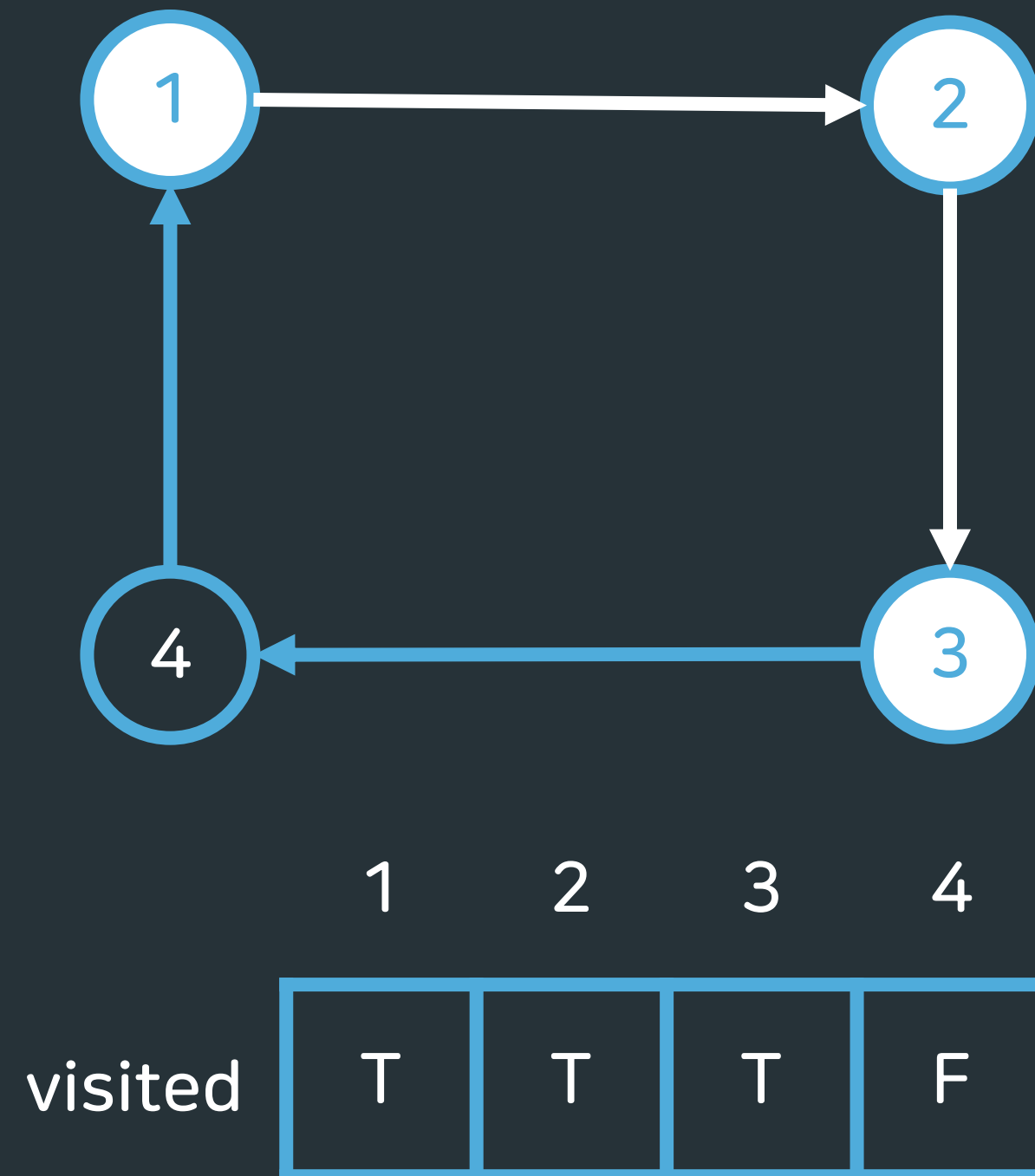
# 사이클이 생성되는 순간



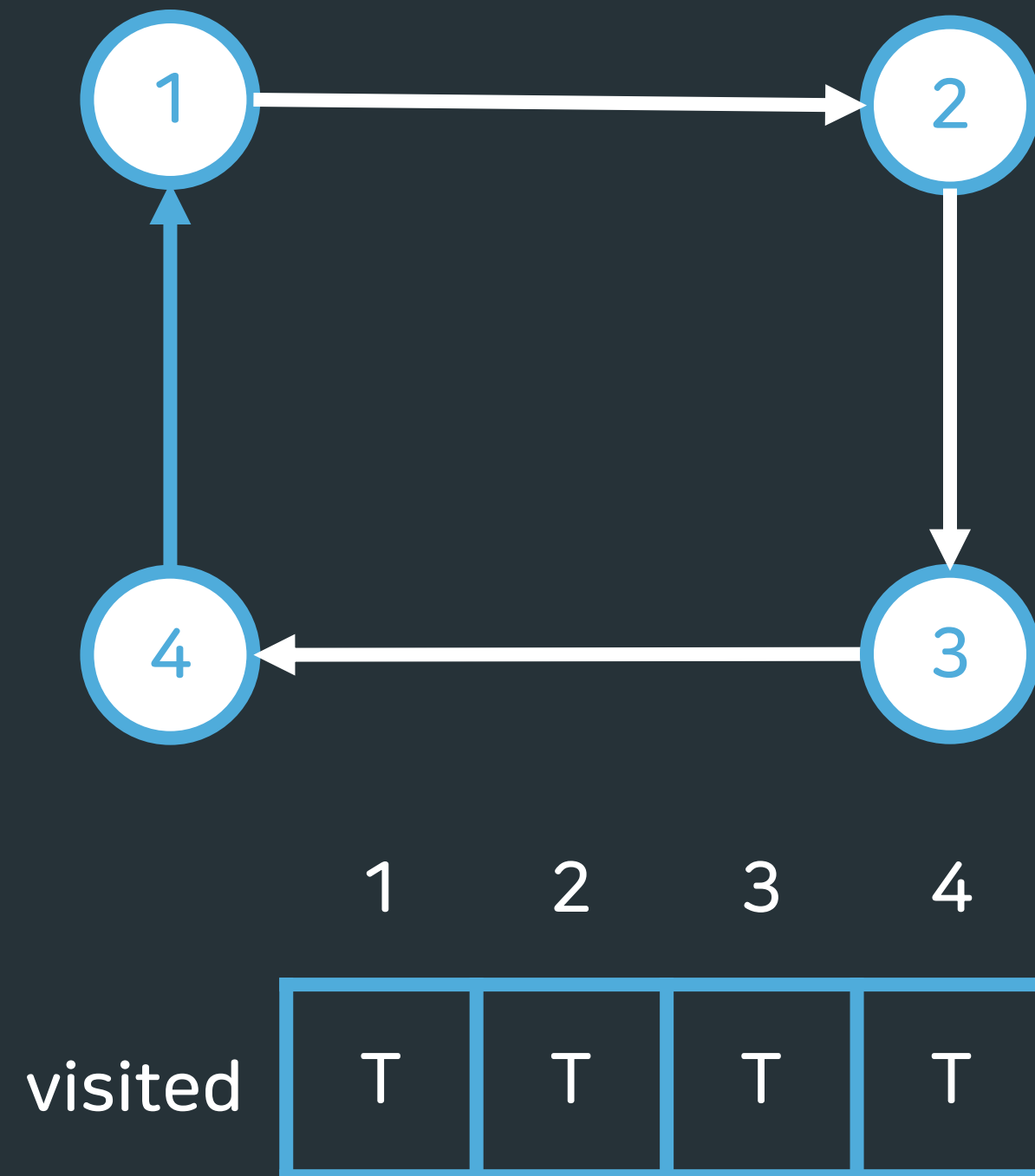
# 사이클이 생성되는 순간



# 사이클이 생성되는 순간

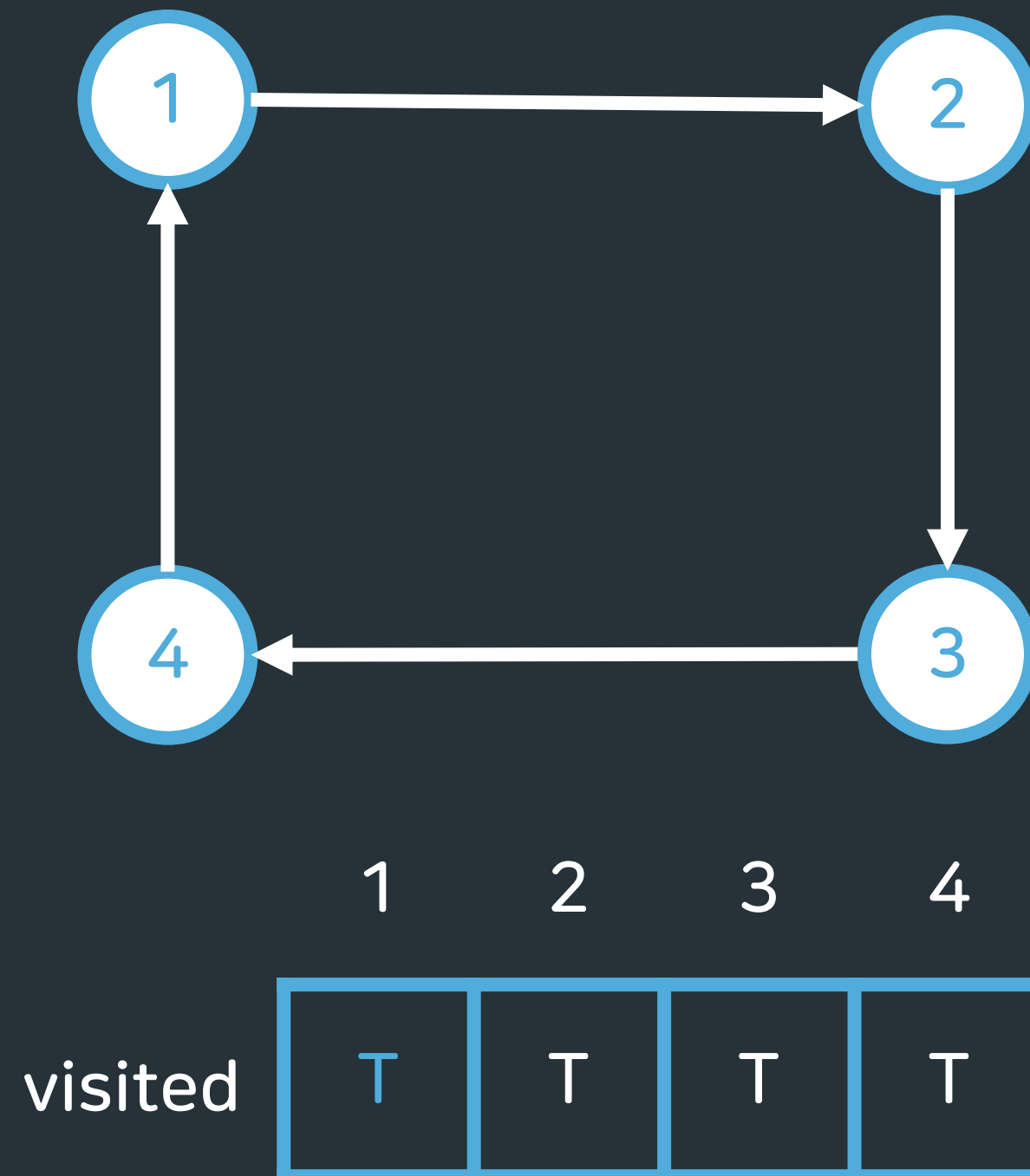


# 사이클이 생성되는 순간

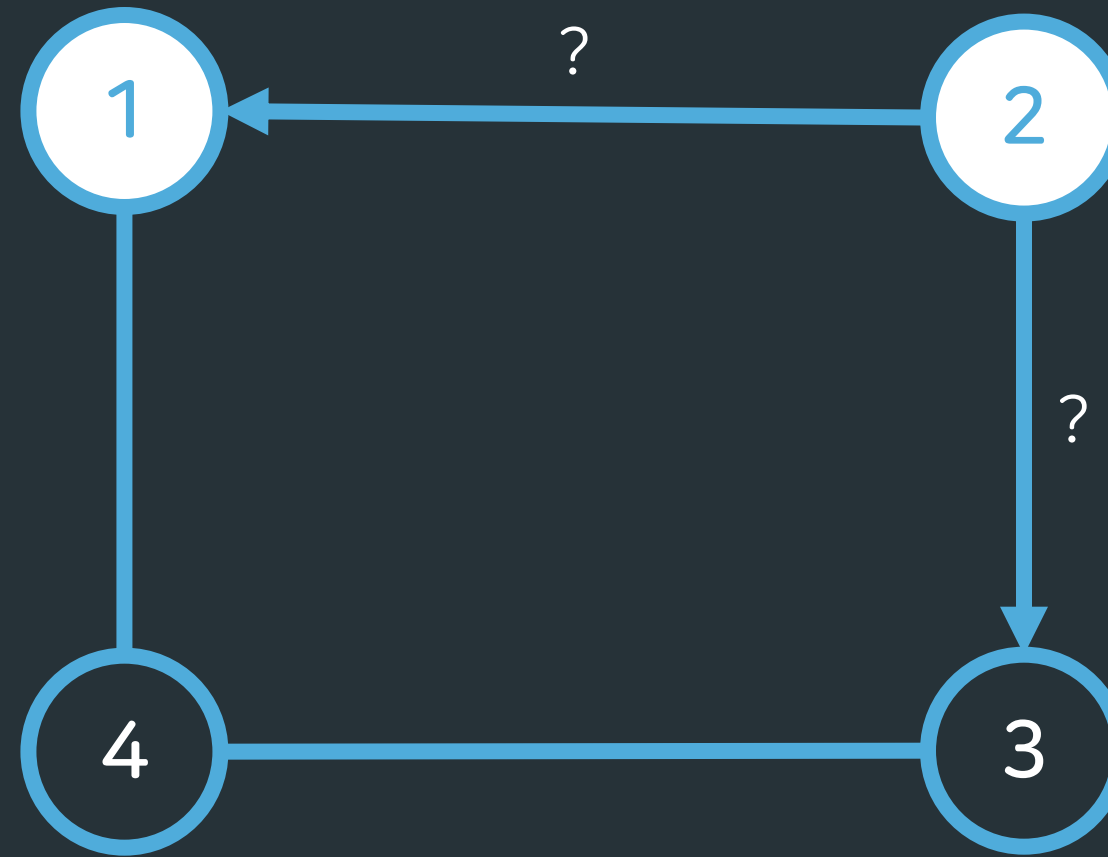


# 사이클이 생성되는 순간

이미 방문한  
1번 정점을 다시 방문  
→ 사이클 생성



## 입력은 무방향 그래프인데...



바로 직전에 탐색한 정점(부모 정점)을 기억해두면  
직전 정점을 다시 탐색해 사이클로 오해하는 일이 없음!  
→ BFS는 직전에 탐색한 정점이 부모 정점이란 보장이 없으므로 어려움

## 정리

- 그래프의 부분집합인 트리
- 그래프와 트리의 차이점을 잘 기억해두기
- 이진 트리와 일반 트리로 나눌 수 있고, 이진 트리는 전위 & 중위 & 후위 순회 가능
- 기본적으로 그래프의 한 종류라서 DFS, BFS 탐색 가능

## 이것도 알아보세요!

- 일반 트리를 이진 트리도 바꿀 수도 있습니다. 어떻게 하면 될까요?
- 우리에게 익숙한 BST에서 파생된 여러 트리들에 대해 알아보세요.  
AVL Tree, Red-Black Tree, B Tree



## 필수

- /<> 3190번 : 뱀 - Gold 4
- /<> 15681번 : 트리와 쿼리 - Gold 5
- /<> 5639번 : 이진 검색 트리 - Gold 5

## 도전

- /<> 1967번 : 트리의 지름 - Gold 4
- /<> 24545번 : Y - Platinum 5

과제제출 마감 ~ 11월 22일 수요일 18:59

추가제출 마감 ~ 11월 23일 목요일 23:59

## /<> 24545번 : Y – Platinum 5

### 문제

- Y-트리의 조건
  1. 4개 이상의 정점과 인접한 정점은 없다.
  2. 인접한 정점의 개수가 3개인 정점은 정확히 하나만 존재한다.
  3. 인접한 정점이 하나뿐인 정점은 정확히 세 개 존재한다.
- Y-트리의 크기는 해당 Y-트리를 이루는 정점의 개수이다.
- 트리가 주어질 때, 정점을 0개 이상 삭제하여 만들 수 있는 가장 큰 Y-트리의 크기를 구하시오.

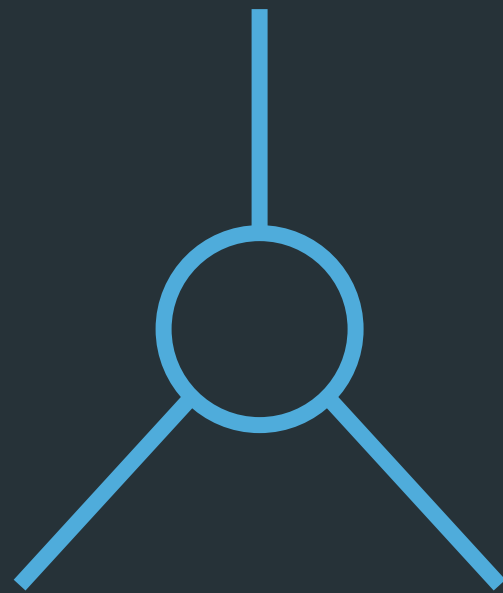
### 제한 사항

- 정점의 개수  $n$ 은  $2 \leq n \leq 100,000$

## Y-트리는 어떤 모양일까요?

1. 4개 이상의 정점과 인접한 정점은 없다.  
→ 한 정점의 간선은 최대 3개!
2. 인접한 정점의 개수가 3개인 정점은 정확히 하나만 존재한다.
3. 인접한 정점이 하나뿐인 정점은 정확히 세 개 존재한다.

### 사용할 수 있는 정점의 종류

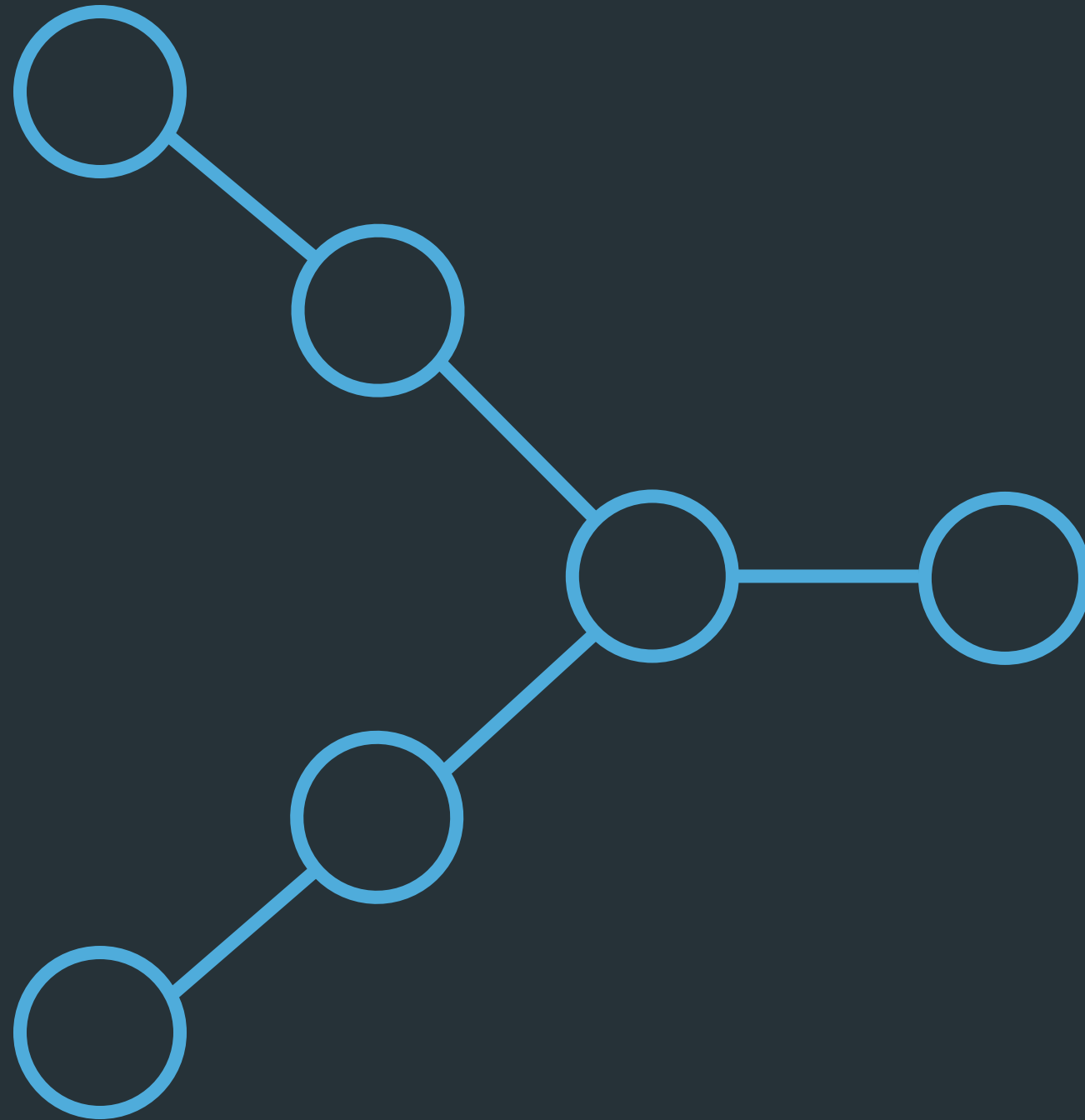


1개



3개

# Y-트리는 어떤 모양일까요?



→ Y 모양!

## Hint

1. 주어진 조건을 만족하는 트리는 어떤 모양일까요?
2. Y-트리의 정점을 세기 위해서 DFS/BFS 중 어떤 것을 활용해야 할까요?
3. 정점의 수가 100,000이에요. 주어진 시간 안에 전부 탐색할 수 있을까요?